TSSI

Programación I

Ing. Pinkas Leonardo

Ing. Mostovoi Alejandro

Año 2020

Índice

Introducción **7**
¿Qué es programar? 7
Pasos para la resolución de un problema 7
Comprender el problema 7
Analizar el problema 7
Diseñar el algoritmo 10
Verificar la solución 10
Codificar 11

Lenguajes de Programación **11**
¿Qué es un lenguaje de programación? 11
Lenguaje máquina 12
Lenguaje de bajo nivel 12
Lenguaje de alto nivel 13

Tipos de programación **13**
Secuencial 13
Estructurada 13
Lógica 14
Orientada a objetos (POO) 14

Programación Estructurada **14**

Variables y Constantes **16**
Variables 16
Constantes 18
Tipos de datos 18
Operadores y Expresiones 19
Asignación 19
Aritméticos 20
Lógicos y Relacionales 20

Algoritmos **25**
Definición 25
Características de un algoritmo 25
Diseño de un algoritmo 26

Representación de un algoritmo **28**
Pseudocódigo 28
Diagrama de Flujo 29

Entradas y Salidas Estándar **30**
Obtención de entrada del usuario. 31
Impresión de salida en pantalla. 31

Contadores y Acumuladores **32**

Estructuras de Control **35**
Estructura de control “Si...Entonces” 35
¿Cómo funciona esta estructura de control? 36
Condiciones múltiples 37
Estructura de control “Según” 38
¿Cómo funciona esta estructura de control? 39
Anidando estructuras 40

Estructuras de Repetición **43**
Estructura de repetición “Mientras...Hacer (While)” 44
¿Cómo funciona esta estructura? 44
Estructura de repetición “Para...Hacer (For)” 47
¿Cómo funciona esta estructura? 47
Cambiando el incremento 50

Funciones y Procedimientos **56**
Desarrollo Top Down 57
Razones para Modularizar 57
Abstracción 58
Ocultamiento de la información 59
Comunicación entre Módulos 59
Tipos de Módulos 60
Función 60
Funciones en C/C++ 60
Invocando funciones 62
Diagramas 64
Ámbito de las variables 64
Variables Globales 65
Variables Locales 66
Procedimientos 68
Parámetros de Salida / Pasaje por Referencia 68
Parámetros de Entrada / Pasaje por Valor 69
Combinando todo 71
Consejos para Modularizar 72

Recursividad 72
Concepto de recursividad: 74

Vectores y Estructuras **78**
Vectores 78
Declaración de un vector 79
Declaración de vectores 80
Acceso a cada elemento 81
Representación gráfica 82
Algoritmos básicos: Cargar Vector 83
Algoritmos básicos: Mostrar Vector 84
Arreglos como parámetros 85
¿Cómo usamos los vectores? 86
Matriz 88
Declaración 88
Acceso a cada elemento 89
Representación gráfica 90
Algoritmos básicos: Cargar Matriz 90
Algoritmos básicos: Mostrar Matriz 91
Matrices como parámetros 92
Estructura de Datos 94
Campos 95
Algoritmos básicos 96
Diferencias con Arreglos 97

Búsqueda **101**
Búsqueda Secuencial 101
Algoritmo 101
Búsqueda por Posición Única Predecible (PUP) 101
Búsqueda binaria 103
¿Cómo funciona la búsqueda binaria? 104

Algoritmos de Ordenamiento **106**
Insertion Sort 107
Burbujeo 109
Burbujeo Mejorado 111

Apareo de Vectores **113**
Apareo de vectores ordenados 113
Apareo de vectores de registros 116

Corte de Control **117**
Corte de Control por múltiples criterios 120
¿Cuándo conviene usarlo? 120

Depuración **123**
Definiciones 123
Bug 123
Falla 123
Depuradores 123

Errores fáciles de corregir **123**

Errores difíciles de corregir **125**

Tipos de Errores **126**
Errores Sintácticos 126
Errores Lógicos 126
Errores de Regresión 126

Depurando con CodeBlocks **127**
Configurar CodeBlocks para depurar 127
Compilar para depurar 128
Establecer un Punto de Interrupción (breakpoint) 128
Observar el valor de una variable 128
Ver la pila de ejecución 129

Tipo Abstracto de Datos (TAD) **131**
El rol de la abstracción 131
Tipo abstracto de dato 131
Ventajas de los TAD 132
Especificación de un TAD 133
Especificación informal 133
Implementación de un TAD en C 134
Archivos de cabecera (parte pública) 134
Implementación de funciones (parte privada) 135
Aplicación de un TAD 135

Bibliografía **136**

Unidad Temática N° 1

~ Conceptos Básicos ~

"Si crees que puedes o crees que no puedes, tienes razón".

- Henry Ford

Introducción

¿Qué es programar?

Programar es planificar, es preparar de antemano las actividades que se van a desarrollar para llevar a cabo una tarea o resolución de problemas.

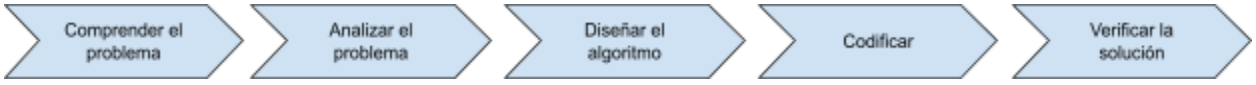
En esta materia aprenderemos a resolver problemas utilizando algoritmos, estructuras de datos y lenguajes de programación.

Ahora bien, ¿Qué es un problema? ¿Cómo llegamos a obtener la solución a dicho problema?

Definición	**¿Qué es un problema?** Podemos decir que un problema es el planteo de una situación a resolver mediante la aplicación de algún método.

El proceso de resolución de un problema con una computadora conduce al desarrollo de un programa y su ejecución en la misma. Si bien este proceso es esencialmente un proceso creativo, podemos considerar una serie de pasos comunes a seguir.

Pasos para la resolución de un problema



Comprender el problema

Este paso, aunque parezca una obviedad, no siempre se realiza de forma correcta y no se puede hallar la solución de algo que no comprendemos. En este paso debemos determinar cuál es el problema planteado. En nuestro caso la especificación o descripción del problema a resolver vendrá dada en forma de enunciado, por lo cual debemos leer la cantidad de veces que sea necesaria dicho enunciado hasta asegurarnos de haber comprendido “cuál es el problema”

Analizar el problema

Un problema consta de varios elementos que debemos identificar en este paso. Es por eso que para poder realizar un buen análisis del problema es conveniente poder responder a las siguientes preguntas:

1. ¿Cuál es el objetivo del problema?

La primer pregunta que nos deberíamos hacer para comenzar a resolver un problema es **"¿Qué quiero obtener?"** al resolver el problema. La respuesta a esta pregunta nos conduce a **identificar el objetivo** del problema. Nuevamente lo que se trata de definir en este punto es el **Qué**.

Tener en cuenta que el objetivo **no nos dice cómo** resolver el problema.
Ejemplo:

> Si mi problema consiste en "ir a Mar del Plata", entonces el objetivo (*el "Qué quiero obtener"*) es llegar a la ciudad de Mar del Plata. Ahora bien, si voy en auto, tren o micro eso sería "el cómo" y no formaría parte del objetivo.

2. ¿Qué datos de entrada se requieren?

Para responder a esta pregunta debemos identificar:

1. **Datos de entrada**: son los datos con los que se debe contar para poder resolver el problema. Por cada uno de estos datos debemos saber:
    a. **Su nombre**: a cada dato de entrada debemos definirle un nombre para poder identificarlo. Este nombre deberá ser representativo para que quede claro qué representa.
    b. **Su característica (Constante o Variable)**: esta clasificación dependerá de si el valor del dato de entrada puede cambiar o no durante la resolución del problema. Si el valor no cambia entonces lo clasificaremos como *Constante*, pero si el valor del dato de entrada puede variar durante la resolución del problema, lo clasificaremos como *Variable*.
    c. **Su tipo de dato**: el tipo de dato nos define cuál es el conjunto de valores posibles y qué tipo de operaciones puedo realizar sobre dicho dato de entrada. Los tipos de datos que manejaremos son:

Tipo de Dato	Valor posible
Entero (Integer)	Número Entero
Real (Float)	Número Real
Caracter (Character)	Un solo símbolo (letra, dígito, signo)
Cadena (String)	Conjunto de caracteres (palabra, frases)
Lógico (Boolean)	Verdadero o Falso

2. **Precondiciones**: las precondiciones indican qué valores podrán tomar los datos de entrada de tipo Variable para que se consideren válidos. Podríamos decir que las precondiciones imponen una restricción sobre los valores definidos por el tipo de dato.

3. ¿Cuál es la salida deseada?

En este punto debemos especificar cuál o cuáles son los resultados esperados y por cada uno definir el tipo de dato.

No es necesario definir nombre ni tampoco clasificarlo en constante o variable ya que siempre se los considera como variable.

4. ¿Lote de datos de prueba para verificar la solución?

En este punto nos centraremos en definir un conjunto de datos que nos permitan verificar la solución del problema. El lote de datos de prueba se compone de una lista de valores para cada uno de los datos de entrada. Podemos representarlo en forma de tabla de valores

Este lote de datos de prueba será utilizado más adelante, cuando hayamos diseñado la solución, para realizar una *Prueba de Escritorio*. Una prueba de escritorio es una herramienta que nos permitirá verificar si el programa produce la salida deseada.

Ejemplo:

"Desarrollar un procedimiento que permita ingresar el nombre de un alumno y las notas de 1er y 2do parcial y devuelva VERDADERO en caso de haber aprobado la materia o FALSO en caso de no aprobar"

Objetivo: saber si el alumno aprobó o no la materia
Datos de entrada:
nombre: Variable de tipo Cadena. Nombre del alumno.
notaParcial1: Variable de tipo Real. Nota del primer parcial
notaParcial2: Variable de tipo Real. Nota del segundo parcial
CANTIDAD_EXAMENES: Constante de tipo Entero cuyo valor es 2

Precondiciones:
notaParcial1 > 0 Y notaParcial1 <= 10
notaParcial2 > 0 Y notaParcial2 <= 10

Datos de salida:
alumno_aprobado: Lógico

Lote de Datos:

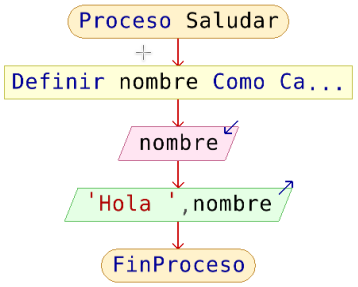
nombre	notaParcial1	notaParcial2	Salida
Leo	5	6	FALSO
Alejandro	8	6	VERDADERO
María	1	10	FALSO

Diseñar el algoritmo

En el paso anterior, nos enfocamos en “Qué” tengo que hacer. En este paso, determinaremos **“Cómo”** resolver el problema. Es decir cómo vamos a procesar los datos de entrada para obtener la salida esperada.

Para diseñar nuestra solución podemos utilizar dos herramientas que son el *pseudocódigo* y el *diagrama de flujo*.

El diagrama de flujo (flowchart) es una representación gráfica de un algoritmo mientras que el pseudocódigo es una herramienta en la que las instrucciones se escriben utilizando palabras en español. Más adelante especificaremos cada una de ellas en detalle.
Ejemplos

Pseudocódigo	Diagrama de flujo
1 Proceso Saludar 2 Definir nombre como Cadena; 3 4 Leer nombre; 5 6 Mostrar "Hola ", nombre; 7 FinProceso	

En la próxima unidad veremos una técnica para poder diseñar un algoritmo.

Verificar la solución

La verificación de una solución es el proceso mediante el cual, con una amplia variedad de datos de entrada, llamados datos de prueba (test) se determina si la solución funciona correctamente o si tiene errores que deberán ser corregidos. El conjunto de datos de prueba debe contemplar

valores normales de entrada, valores extremos de entrada que permitan verificar los límites del programa y valores que permitan comprobar aspectos especiales del programa.

En esta instancia contamos con la *Prueba de Escritorio* como herramienta para verificar la solución. Para utilizar esta herramienta desarrollaremos una tabla con los valores que van tomando los datos de entrada y los datos de salida a por cada instrucción de nuestra solución.

Paso	Entrada_1	...	Entrada_N	Salida_1	...	Salida_N
1						
...						
N						

Codificar

La codificación consiste en traducir la solución diseñada en el paso anterior utilizando un *lenguaje de programación*. Dado que el diseño de una solución es independiente del *lenguaje de programación* a utilizar, debería poder ser escrito con igual facilidad en un lenguaje u otro.

Definición	**¿Qué es un lenguaje de programación?** Es un conjunto de instrucciones (acciones u operaciones) que nos permiten crear programas a ser ejecutados por una computadora.

Lenguajes de Programación

¿Qué es un lenguaje de programación?

Como mencionamos anteriormente, un lenguaje de programación es un conjunto de instrucciones (acciones u operaciones) que nos permiten crear programas a ser ejecutados por una computadora. Este conjunto de reglas nos permiten escribir instrucciones en un “idioma” que puede ser traducido a un lenguaje comprensible para la computadora.
Al lenguaje que comprende la computadora lo llamamos *lenguaje máquina*.

Como todo lenguaje, un lenguaje de programación, tiene su parte sintáctica y su parte semántica. Esto implica que todos los lenguajes de programación poseen reglas sobre cómo deben ser escritas las sentencias y de qué forma (*sintaxis*).

Todo lenguaje de programación posee un conjunto básico de instrucciones comunes que son:

- Instrucciones de entrada/salida. Permiten la transferencia de datos desde y hacia la computadora.
- Instrucciones de cálculo. Permiten realizar operaciones aritméticas.
- Instrucciones de control. Permiten modificar la secuencia de la ejecución del programa.

A su vez, los lenguajes de programación pueden clasificarse en 2 grandes grupos:

- lenguajes de bajo nivel
- lenguajes de alto nivel.

Lenguaje máquina

Es el lenguaje que entiende la máquina sin necesidad de traducciones. El problema es que desarrollar programas utilizando este lenguaje resulta una tarea sumamente difícil para una persona ya que sus instrucciones son secuencias de 0 (ceros) y 1 (unos), tales como 11110000, 011100110, etc, que son muy difíciles de recordar y manipular.

Ejemplo:

```
Mover el contenido del registro 4 al registro 8, se podía expresar como

4048 o bien 0010 0000 0010 1000
```

En consecuencia se necesitan lenguajes de programación “amigables” con las personas, que permitan escribir programas con mayor facilidad.

Lenguaje de bajo nivel

También llamados *ensambladores*, son aquellos lenguajes cuyas sentencias están formadas por códigos nemotécnicos (abreviaturas de palabras inglesas), por lo que sus instrucciones son mucho más comprensibles y fáciles de recordar que los lenguajes de máquina.

Algo a tener en cuenta es que estos lenguajes son dependientes de la arquitectura de cada procesador debido a que cada procesador ofrece su propio conjunto de instrucciones.

Ejemplo:

```
Mover el contenido del registro 4 al registro 8, se podía expresar como

MOV R5, R6
```

Un programa escrito utilizando algún lenguaje de bajo nivel, requiere de un ensamblador (assembler) que lo convierta a lenguaje máquina.

Lenguaje de alto nivel

Son aquellos cuyas instrucciones están formadas por palabras similares a las de los lenguajes naturales (comúnmente utilizado por humanos). Estos lenguajes son independientes del procesador y por lo tanto, resulta mucho más sencillo escribir un programa en un lenguaje de alto nivel para luego ser traducido a lenguaje máquina.

Algunos de los lenguajes de alto nivel más conocidos son C, C++, C#, Java, PHP, Python.

Más allá de que las instrucciones de estos lenguajes son fáciles de recordar y manipular, requieren ser traducidos a código máquina. Para realizar la traducción de lenguaje de alto nivel a lenguaje máquina se utilizan programas llamados *compiladores* y cada lenguaje posee el suyo, es decir que C tendrá su propio compilador, Java tendrá el suyo, etc.

Tipos de programación

No sólo existen distintos tipos de lenguajes de programación sino que también existen distintas formas de programar, cada uno con sus características. Los distintos tipos de programación son:

Secuencial

Este tipo de programación se basa en la construcción de programas cuyas sentencias se escriben y ejecutan de manera secuencial. Se utiliza la instrucción ***goto*** para realizar una bifurcación (división) en el flujo normal de ejecución del programa.

Estructurada

Este tipo de programación se basa principalmente en la modularidad de los programas. Esto quiere decir que cada programa se compone de módulos más pequeños donde cada uno realiza una tarea específica.

En otras palabras, lo que se trata de hacer es de dividir al problema en problemas más pequeños para poder alcanzar soluciones más simples.

Este tipo de programación define 3 reglas:

- El programa debe tener un diseño modular.
- Cada módulo es diseñado de modo descendente.
- Cada módulo es codificado utilizando las 3 estructuras de control básicas: secuencia, selección y repetición (no está permitido el uso de instrucciones ***goto***).

Lógica

La programación lógica es una forma de programar donde se deben definir hechos y relaciones entre los mismos, que conforman la llamada base de conocimiento del programa. Se utiliza el principio de razonamiento lógico para responder a las preguntas realizadas. Se basa en la lógica formal y en el cálculo de predicados de primer orden.

Orientada a objetos (POO)

Este enfoque se basa o guarda cierta analogía con la vida real. El desarrollo de software se basa en el diseño y construcción de objetos que interactúan entre sí para resolver el problema planteado. Cada objeto se compone de datos y acciones que manipulan esos datos.

Programación Estructurada

Como mencionamos anteriormente, la programación estructurada significa escribir un programa de acuerdo a las siguientes reglas:

- El programa debe tener un diseño modular.
- Cada módulo es diseñado de modo descendente.
- Cada módulo es codificado utilizando las 3 estructuras de control básicas: secuencia, selección y repetición (no está permitido el uso de instrucciones ***goto***).

La programación estructurada utiliza un número limitado de estructuras de control que minimizan la complejidad de los programas y por consiguiente ayuda a reducir los errores. Esto hace también que los programas sean más fáciles de escribir, leer y verificar, en otras palabras ayuda a la mejorar la mantenibilidad del código.

La programación estructurada incorpora lo que llamamos *estructuras básicas de control*, que nos permiten especificar y definir el orden en que se ejecutarán las instrucciones de un algoritmo. Estas estructuras son fundamentales en los lenguajes de programación y en el diseño de algoritmos. Las 3 estructuras básicas son:

- sentencia
- estructuras de control
- estructuras de repetición

Unidad Temática N°2

~ Variables y Operadores ~

“Un nombre largo y descriptivo es mejor que un nombre corto y enigmático. Un nombre largo y descriptivo es mejor que un comentario largo y descriptivo ".

Robert C. Martin

Variables y Constantes

Variables

Previamente, cuando hablamos sobre cómo analizar un problema y definir sus datos de entrada y salida, mencionamos que algunos datos de entrada se clasifican como *Variables* y otros como *Constantes*.

Ahora bien ¿Qué es una Variable?

Una variable es una posición, un espacio, de memoria (memoria RAM) a la cual desde nuestro programa le asociamos un nombre para poder accederla, leer y/o modificar su valor.

¿Y para qué me sirve una variable? Bien, una variable la podemos usar para guardar valores que se ingresan al programa como puede ser el nombre de una persona, la edad; para guardar resultados de cálculos que deba realizar el programa como puede ser un promedio, el resultado de una operación aritmética como una suma o una resta, etc.

Toda variable tiene un *tipo de dato* y un nombre o identificador. La persona que desarrolla un programa debe procurar que este nombre sea lo más representativo posible para facilitar el uso de la variable y la lectura del código.

¿Cómo definimos el nombre de una variable?

Cada vez que necesitemos utilizar el valor que guarda una variable, o bien asignarle un valor, lo haremos a través de su nombre. Ahora bien, para definir el nombre de una variable debemos seguir ciertas reglas que a continuación mencionamos:

- ✓ Debe comenzar con una letra (cualquiera entre la ‘a’ y la ‘z’) o el caracter _ (guión bajo).
- ✓ Luego del primer caraceter **sólo** puede contener letras (mayúsculas y/o minúsculas), números o _ (guión bajo).
- ✗ No puede contener espacios en blanco, o cualquier otro caracter que no sean letras (mayúsculas y/o minúsculas), números o _ (guión bajo).

Basándonos en las 3 reglas mencionadas, se suele utilizar una convención llamada *“camelCase”* que es un estilo de escritura que utilizamos cuando el nombre de nuestra variable está formado por varias palabras. En este estilo de escritura la primer letra del nombre de la variable será siempre una letra minúscula y las siguientes primeras letras de las otras palabras serán una letra mayúscula.

Veamos algunos ejemplos

Formas correctas de definir el nombre de una variable	Formas incorrectas de definir el nombre de una variable
✓ `Definir _salario como Real;` ✓ `Definir nombre_alumno como Cadena;` ✓ `Definir letra_4 como Caracter;` Ahora algunos ejemplos utilizando camelCase: ✓ `Definir cuentaBancaria como Entero;` ✓ `Definir nombreAlumno como Cadena;` ✓ `Definir esVocal como Booleano;`	✗ `Definir el salario como Real;` ✗ `Definir nombre alumno como Cadena;` ✗ `Definir 4_letra como Caracter;` ✗ `Definir esVocal? como Booleano:` ✗ `Definir legajo-empleado como Entero;`

¿Por qué una variable tiene que tener un tipo de dato?

El tipo de dato de una variable define 2 cosas, en primer lugar define el rango de valores posibles que puede guardar una variable, así si una variable la defino como de *tipo Entero* sólo podrá almacenar números enteros, por ejemplo: -1040, -2, 0, 10, 20, 130...
En segundo lugar, el tipo de dato de una variable, define qué cosas puedo hacer con esa variable, es decir, qué operaciones puedo ejecutar sobre esa variable. De nuevo si una variable la defino como de *tipo Entero* sólo voy a poder realizar operaciones que estén definidas para los números enteros como ser suma, resta, multiplicación, división, etc. Pero si tuviese una variable de *tipo Cadena* claramente no voy a poder ejecutar una división de 2 cadenas porque la operación no está definida para el tipo de dato Cadena.

Toda variable debe ser definida antes de poder ser utilizada, es decir antes de poder asignarle un valor o de intentar leer el valor que tiene, la variable debe ser definida.

Veamos algunos ejemplos

Ejemplo correcto	Ejemplo incorrecto

`// Define la variable edad de tipo Entero`
`Definir edad como Entero;`
 	
`// Asigna un valor a la variable edad`
`edad = 18;`
	`// Trata de asignar un valor a la variable edad, pero la variable edad no fue definida.`
	`edad = 18;`
	`----------`

```
// Define la variable nombre de tipo Cadena
Definir nombre como Cadena;

// Asigna un valor a la variable nombre
nombre = “Pepe”;
```

```
Definir resultado como Entero;
Definir nombre como Cadena;

nombre = “Pepe”;

// Trata de realizar una multiplicación con
una variable de tipo cadena
resultado = nombre * 2;
```

Constantes

Al igual que las variables, una constante es una posición, un espacio, de memoria (memoria RAM) a la cual desde nuestro programa le asociamos un nombre para poder accederla y leer su valor pero lo que nunca vamos a poder hacer es cambiarle el valor que le asignemos inicialmente.

Las constantes también poseen un tipo de dato y debemos definirle un nombre siguiendo las mismas reglas que utilizamos para definir el nombre de una variable.

Tipos de datos

El concepto de tipo de dato no es propio de la programación estructurada, sino que aplica a cualquier tipo de programación que utilicemos.

Los lenguajes de programación permiten trabajar con tipos de datos básicos (primitivos) como los que mencionamos al describir cómo analizar un problema.

Los tipos de datos los utilizamos para definir datos de entrada (variables, constantes) y datos de salida.

Tipo de Dato	Valor posible	Ejemplo
Entero (Integer)	Número Entero	Definir **num** como Entero; La variable **num** puede guardar valores como -10, 3, 9, 100. La variable **num** no puede guardar valores como 3,14
Real (Float)	Número Real	Definir **precio** como Real; La variable **precio** puede guardar valores como -10, 3, 3.5, 9, 9.9, 100, 100.2.
Caracter (Character)	Un solo símbolo (letra, dígito, signo)	Definir **letra** como Caracter; La variable **letra** puede guardar valores como “a” o signo “$”.

		La variable **letra** no puede guardar valores como “Pepe”
Cadena (String)	Conjunto de caracteres (palabra, frases)	Definir **nombre** como Cadena; La variable **nombre** puede guardar valores como “Pepe” y también como “a”, “$”
Lógico (Boolean)	Verdadero o Falso	Definir **esNumero** como Lógico; La variable **esNumero** puede guardar valores como Verdadero o Falso únicamente.

Más adelante veremos que tomando como base los tipos de datos primitivos podemos crear tipos de datos que llamaremos *tipo de dato definido por el usuario*.

Operadores y Expresiones

Los operadores nos permiten realizar cálculos aritméticos, asignaciones, evaluar relaciones y expresiones lógicas

Asignación

El operador de asignación es el caracter “=” y lo utilizamos para asignar un valor a una variable.

Ejemplo:

```
Definir numero Como Entero;  // Declara la variable numero

numero = 4; // y asigna el valor 4.
```

¿Qué es una asignación? En programación, usamos el término asignación para indicar que estamos dando valor a una variable.

Para realizar una asignación utilizamos el signo “=” y una sintaxis (forma de escribir) similar a la matemática donde ponemos a izquierda del signo igual la variable y a derecha del signo igual el valor o expresión. De la siguiente manera:

“variable” = “valor a asignar”;

Se debe tener en cuenta que la variable y el valor a asignar deben ser de tipos “compatibles” o asignables

Ejemplo:

```
Definir numero como Entero; // Define la variable numero de tipo Entero
numero = 3; // Le asigna el número 3 a la variable numero
```

```
Definir promedio como Real; // Define la variable promedio de tipo Real
promedio = (7 + 8) / 2
```

Aritméticos

Las operaciones aritméticas se llevan a cabo utilizando los operadores **+, -, *, /** que se corresponden con las operaciones **suma, resta, multiplicación y división**.

También existe el operador **%** que nos permite obtener el resto de una división.

Estos operadores sólo se pueden usar entre variables, constantes o expresiones que sean de tipo numérico: Entero o Real.

Ejemplos:

```
Definir suma Como Entero;  // Declara la variable suma
suma = 4 + 5;              // A la variable suma le asigna el resultado de sumar 4 + 5.

Definir resta Como Entero; // Declara la variable resta
resta = 3 - 2;           // A la variable resta le asigna el resultado de restar 3 - 2.

Definir producto Como Entero; // Declara la variable producto
producto = 4 * 5; // A la variable producto le asigna el resultado de multiplicar 4 * 5

Definir cociente Como Entero; // Declara la variable cociente

// A la variable cociente le asigna el resultado de dividir 19 / 5. Esto es 3
cociente = 19 / 5;

Definir resto Como Entero; // Declara la variable resto

// A la variable resto le asigna el resto de dividir 19 / 5. Esto es 4
cociente = 19 % 5;
```

Lógicos y Relacionales

Tanto los operadores relacionales como los lógicos se presentan, generalmente, en expresiones que devuelven un valor Booleano, es decir un valor que puede ser Verdadero (true) o Falso (false).

Los operadores relacionales son utilizados para realizar comparaciones entre valores de 2 o más variables mientras que los operadores lógicos nos sirven de soporte para armar expresiones que vamos a relacionar.

Estos operadores sólo se pueden usar entre variables, constantes o expresiones que sean del tipo Lógico

Operadores Relacionales

Operador	Acción
>	mayor que
<	menor que
>=	mayor o igual que
<=	menor o igual que
==	igual a
!=	distinto de (no igual)

Operadores Lógicos

Operador	Acción
Y	conjunción
O	disyunción
NO	negación

Ejemplos:

```
Definir resultado Como Logico; // Declara la variable resultado;
// La variable resultado guardará el valor Falso (false) ya que 4 no es mayor que 5
resultado = 4 > 5;
// La variable resultado guardará el valor Verdadero (true) ya que 4 es menor (o
// igual) que 5
resultado = 4 <= 5;
// La variable resultado guardará el valor Verdadero (true) ya que 4 es mayor que 3
// Y 2 es mayor (o igual) a 2
resultado = 4 > 3 Y 2 >= 2;
```

A continuación mostramos las operaciones lógicas y cuál es su tabla de verdad

p	q	p Y q	p O q	NO q

F	F	F	F	V
F	V	F	V	F
V	F	F	V	V
V	V	V	V	F

Expresiones Combinadas

Las operaciones y expresiones mencionadas en los puntos anteriores pueden combinarse de forma que se realicen varias operaciones en una misma sentencia. Al escribir una expresión que combine varios operadores se debe tener en cuenta las reglas de precedencia de los mismos, la siguiente tabla muestra el orden de precedencia siendo los de mayor nivel los de más arriba:

- multiplicación y división: * / %
- suma y resta: + -
- operadores relacionales: < > <= =>
- equivalencia: == !=
- conjunción: Y
- disyunción: O
- operadores de asignación: =

Cabe destacar que **lo último que se hará en una sentencia es realizar la asignación**, por eso está al final de la tabla. Esto tiene sentido dado que antes de poder asignar un valor se debe resolver toda la expresión a calcular.

Veamos unos ejemplos de cómo actúa esta regla.

```
j = 1 + 3 * 4; // resultado j = 13
```

Desde que aprendimos aritmética básica, conocemos la regla que nos obliga a calcular la multiplicación antes de una suma.

```
j = 1 + 3 - 4;  resultado j= 0;
```

Si todos los operadores tienen un nivel idéntico de precedencia **se evalúa la expresión de izquierda a derecha**.

En caso que se desee forzar una precedencia se deben utilizar paréntesis

```
bol = (8 + 10) / 2 > 6;  resultado bol= Verdadero;
```

Primero resuelve la suma por el uso de los paréntesis, luego la división y luego resuelve la comparación con 6, resultando en verdadero

Como en matemáticas, podemos anidar los paréntesis. Se comenzará a evaluar los internos hasta llegar a los externos.

Cabe agregar que los paréntesis no disminuyen el rendimiento de los programas. Por lo tanto, agregar paréntesis no afecta negativamente al programa.

```
k = ((12 - 2) * ( 21 - 11)) / ((1+1)*(15-10)) + 1 ;
```

Unidad Temática N°3

~ Algoritmos ~

"Hay dos formas de diseñar software: una es hacer que sea tan simple que obviamente no haya deficiencias, y la otra es hacer que sea tan complicado que no haya deficiencias obvias. El primer método es mucho más difícil."

- C. A. R. Hoare

Algoritmos

Definición

Antes de adentrarnos en la definición de qué es un algoritmo, cabe mencionar que una persona que se desempeña como programador es antes una persona que resuelve problemas, por lo que para llegar a ser un programador eficaz se necesita aprender a resolver problemas de un modo metódico y sistemático. Previamente hablamos sobre la metodología necesaria para resolver un problema mediante el uso de una computadora. El eje central de esta metodología es el concepto de *algoritmo.*

Ahora sí y sin más preámbulo, la definición de **algoritmo** es: un método para resolver un problema.

Si bien la anterior definición es la más simple que podemos encontrar, la definición de algoritmo que más se aproxima a nuestro propósito es la siguiente:

Definición	Secuencia finita de instrucciones, reglas o pasos que describen en forma precisa las operaciones que una computadora debe realizar para llevar a cabo una tarea en tiempo finito [Knuth, 1968].

Los algoritmos son independientes tanto del lenguaje de programación en el que se expresan como de la computadora que los ejecuta, así un algoritmo puede ser escrito en diferentes lenguajes y ser ejecutado por distintas máquinas y sin embargo el algoritmo será el mismo.

Tratando de hacer una analogía, una receta de cocina se puede escribir en diferentes idiomas (español, inglés, portugués), pero los pasos de la receta seguirán siendo los mismos.

Tener en cuenta que los algoritmos son mucho más importantes que los lenguajes de programación que se utilizan para expresarlos y que las computadoras en las que son ejecutados; es por esto que el proceso de diseño de un algoritmo es uno de los aspectos más importantes y nos va a demandar creatividad, conocimientos y técnicas para lograr una buena solución.

Características de un algoritmo

Las características fundamentales de todo algoritmo son:

- Debe ser *preciso*. Indicando el orden de realización de cada paso.

- Debe estar bien definido y no ser ambiguo. Si se sigue un algoritmo 2 veces debemos alcanzar el mismo resultado.
- Debe ser finito. Debe tener un número finito de pasos. Debe terminar en algún momento.

La definición de todo algoritmo debe describir 3 partes:

1. Los datos de entrada (volver a leer la Unidad 1)
2. El proceso: los pasos a ejecutar
3. La salida esperada

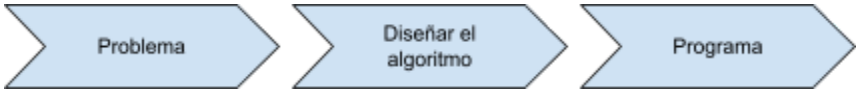
Ejemplo:
Problema: “Un cliente realiza un pedido a una fábrica. La fábrica examina los datos del cliente para saber si tiene deuda o no. Si el cliente no posee deuda, la empresa acepta el pedido, en caso contrario lo rechaza”

Algoritmo

```
1. Inicio
2. Leer el pedido
3. Analizar estado del cliente
4. Si el cliente no es deudor
       a. escribir acepta el pedido
5. Caso contrario
       a. escribir rechaza el pedido.
6. Fin.
```

Diseño de un algoritmo

La resolución de un problema demanda el diseño de un algoritmo que resuelva el problema propuesto.



Cualquier problema se puede resolver más eficazmente cuando se lo divide en problemas más pequeños que sean obviamente más fáciles de solucionar. A esta técnica se la conoce como “*divide y vencerás”*.

La descomposición de un problema en subproblemas más simples y a su vez la división de estos subproblemas en otros más simples que puedan ser implementados para su solución en la computadora se la denomina **diseño descendente**.

Normalmente, lo que suele suceder es que, nuestro primer planteo del algoritmo tendrá pocos pasos (que agrupen grandes tareas) y esté incompleto o sea ambiguo, por lo que debemos

tomar cada uno de esos pasos y volver a dividirlos, en pasos más pequeños y detallados, hasta alcanzar un grado de descomposición de las tareas que se asemeje a instrucciones que puedan ser volcadas a una computadora. A este último paso lo denominamos *refinamiento*.

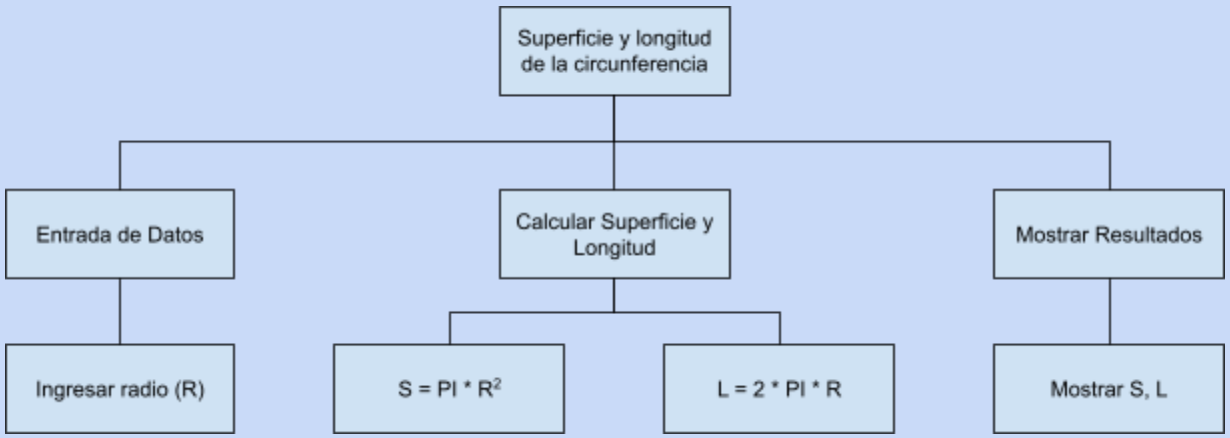
Entonces, el algoritmo para el diseño de un algoritmo podría ser algo como:

```
1. Inicio.
2. Descomponer el problema en problemas menores
3. Si tengo un grado de refinamiento que me permite volcarlo a la computadora entonces:
       a.   entonces => Tengo mi algoritmo!! :). Fin.
4. Sino
       a.  volver al punto 2.
5. Fin.
```

Vamos a verlo con un ejemplo sencillo:

Problema: “Dado el radio de una circunferencia, mostrar la superficie y el perímetro de la misma”

Diseño del Algoritmo


Superficie y longitud de la circunferencia
Entrada de Datos
Calcular Superficie y Longitud
Mostrar Resultados
Ingresar radio (R)
S = PI * R2
L = 2 * PI * R
Mostrar S, L

En este ejemplo simple, vemos como en un primer paso, dividimos el problema en 3 (grandes) tareas: “Entrada de Datos”, “Calcular Superficie y Longitud”, “Mostrar Resultados”. Pero estas tareas siguen siendo muy grandes y complejas como para poder implementarlas, por lo que nos conviene realizar una iteración más, un refinamiento de cada una de estas tareas para poder lograr un nivel de simpleza en cada una que nos permita volcarlo a la computadora. Es ahí donde surge el segundo nivel.

Representación de un algoritmo

Previamente mencionamos que lo más importante es lograr diseñar el algoritmo que nos resuelve el problema planteado, debido a que puede tener varias representaciones según el lenguaje elegido, la computadora donde se ejecute, etc. Bien en este punto vamos a describir 2 herramientas que nos permitirán representar un algoritmo de forma precisa sin preocuparnos, todavía, por la elección de un lenguaje de programación.

Estas herramientas son el pseudocódigo y el diagrama de flujo.

Pseudocódigo

El pseudocódigo, podríamos decir que es un lenguaje (pero no un lenguaje de programación) para especificar un algoritmo. Este lenguaje hace que el paso a un lenguaje de programación sea relativamente sencillo.

El pseudocódigo nació como un medio para representar las estructuras de control de la programación estructurada (cosas que vamos a ver en las próximas unidades) . Se lo considera un primer borrador ya que el pseudocódigo debe ser traducido a un lenguaje de programación específico. El pseudocódigo no puede ser ejecutado por una computadora.

La ventaja que nos proporciona el uso de esta herramienta, es poder concentrarnos en cómo resolver el problema y no preocuparnos por las reglas del lenguaje de programación. Otra de las ventajas es que al ser escrito en el idioma nativo de la persona que programa puede ser utilizado como una herramienta de comunicación para con otros programadores. Es fácil de modificar.

Todo algoritmo en pseudocódigo tiene la siguiente estructura general:

```
Algoritmo MiPrimerAlgoritmo
     acción 1;
     acción 2;
     ...
     acción n;
FinAlgoritmo
```

Comienza con la palabra clave *Algoritmo* (o alternativamente *Proceso*, son sinónimos) seguida del nombre del programa, luego le sigue una secuencia de instrucciones y finaliza con la palabra *FinAlgoritmo* (o *FinProceso*). Una secuencia de instrucciones es una lista de una o más instrucciones y/o estructuras de control.

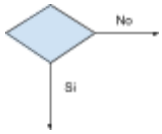
Ejemplo

```
Proceso Suma

        Definir A,B,C como Reales;

        // para cargar un dato, se le muestra un mensaje al usuario
        // con la instrucción Escribir, y luego se lee el dato en
        // una variable (A para el primero, B para el segundo) con
        // la instrucción Leer

        Escribir "Ingrese el primer numero:";
        Leer A;

        Escribir "Ingrese el segundo numero:";
        Leer B;

        // ahora se calcula la suma y se guarda el resultado en la
        // variable C mediante la asignación (=)

        C = A+B;

        // finalmente, se muestra el resultado, precedido de un
        // mensaje para avisar al usuario, todo en una sola
        // instrucción Escribir

        Escribir "El resultado es: ",C;

FinProceso
```

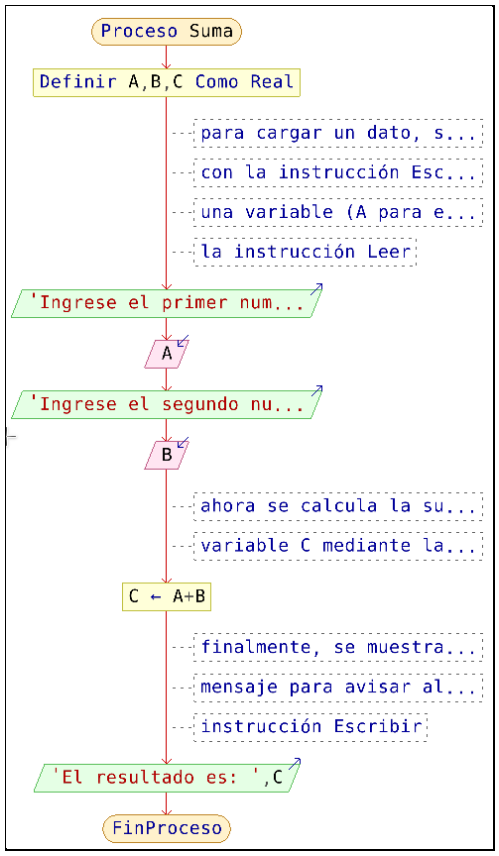
Diagrama de Flujo

Un diagrama de flujo es un diagrama que utiliza símbolos estandarizados que nos permiten representar los pasos de un algoritmo. Estos símbolos se encuentran unidos por líneas o flechas, denominadas líneas de flujo, que indican la secuencia en que se debe ejecutar.

Símbolo	Descripción
	Terminal. Permite representar el principio y el fin del algoritmo.
	Entrada / Salida. Utilizado para ingreso de datos y también para mostrar la salida de información.
	Proceso. Cualquier tipo de operación que pueda originar cambio de valor.

	Decisión. Evalúa una expresión lógica y en función de su resultado Verdadero o Falso, determina el camino a seguir.
	Decisión múltiple. De acuerdo al resultado de comparación se seguirá una opción u otra.
	Conector a otra página.

Ejemplo



Entradas y Salidas Estándar

Los programas interactúan con el exterior a través de los datos de entrada y salida. Los datos de entrada, en su mayoría, serán ingresados por un usuario a través del teclado, mouse, pantalla

táctil mientras que las salidas será información que el programa ofrecerá al usuario a través de la pantalla o en su defecto una impresora.

Obtención de entrada del usuario.

Para obtener los datos de entrada que nos proporcionará el usuario, debemos en primera instancia definir todas las variables necesarias para poder almacenar dichos datos. En segundo lugar debemos ejecutar una sentencia, una instrucción, que nos permita pedirle al usuario cada uno de los datos.

Veamos un ejemplo de esto en pseudocódigo:

Problema: “Pedirle al usuario que ingrese su nombre y mostrar un mensaje por pantalla saludándolo.”

```
1  Proceso Saludar
2     Definir nombreUsuario como Cadena;
3     Mostrar “Ingrese su nombre por favor: “;
4     Leer nombreUsuario;
5     Mostrar “Hola “, nombreUsuario;
6  FinProceso
```

Del ejemplo anterior podemos destacar algunas cuestiones importantes:

- En primer lugar en la línea #2 estamos definiendo la variable *nombreUsuario* para poder guardar los datos que el usuario vaya a ingresar
- En la línea #3 estamos utilizando la instrucción *Mostrar* que nos permite imprimir un mensaje en la pantalla.
- En la línea #4 estamos utilizando la instrucción *Leer* que nos permite obtener los datos ingresados por el usuario y guardarlos en la variable *nombreUsuario*.

Impresión de salida en pantalla.

Cuando desde el programa necesitamos mostrarle un mensaje al usuario o bien presentar los resultados de nuestro algoritmo, vamos a utilizar la pantalla de la computadora o como dijimos al comienzo una impresora.

Para poder mostrar algo por pantalla debemos ejecutar una sentencia que imprima por pantalla el mensaje y/o dato en cuestión.

Veamos un ejemplo de esto en pseudocódigo:

Problema: “Pedirle al usuario que ingrese 2 números enteros y mostrar su suma.”

```
1  Proceso Sumar
2     Definir numero1 como Entero;
3     Definir numero2 como Entero;
4     Definir suma como Entero;
5
6     Mostrar “Ingrese el primer número por favor: “;
7     Leer numero1;
8
9     Mostrar “Ingrese el segundo número por favor: “;
10    Leer numero2;
11
12    suma = numero1 + numero2;
13
14    Mostrar “La suma es: “, suma;
15 FinProceso
```

Del ejemplo podemos ver (además de la definición de variables necesaria para poder guardar los datos que ingresa el usuario) que la instrucción *Mostrar* se puede utilizar tanto para imprimir un mensaje por pantalla, como en las líneas #6 y #9, como para imprimir un mensaje y valores como en la línea #14.

Contadores y Acumuladores

¿Qué es un contador?

En primer lugar es una variable, de tipo Entero, que como su nombre lo indica su función es contar cosas.

¿Y un acumulador?

También es una variable, sólo que su tipo dependerá de qué valor necesito acumular. Por ejemplo si necesito acumular valores de tipo Real, mi acumulador deberá ser de tipo Real. Un acumulador irá sumando valores a los valores que tenía guardado previamente.

Veamos un ejemplo para clarificar:

Problema: “Obtener el promedio de notas de un alumno y mostrarlo por pantalla”

```
1  Proceso Promedio
2
3     Definir nota_1 como Real;
4     Definir nota_2 como Real;
5     Definir nota_3 como Real;
6
7     Definir sumaParcial como Real; // Defino mi acumulador
8
```

```
 9      Definir cantidadDeNotas como Entero; // Defino mi contador
10
11      Mostrar “Ingrese la primer nota: “;
12      Leer nota_1;
13      sumaParcial = sumaParcial + nota_1; // acumulo el valor de la nota 1
14      cantidadDeNotas = cantidadDeNotas + 1; // cuento una nota más
15
16
17      Mostrar “Ingrese la segunda nota: “;
18      Leer nota_2;
19      sumaParcial = sumaParcial + nota_2; // acumulo el valor de la nota 2
20      cantidadDeNotas = cantidadDeNotas + 1;  // cuento una nota más
21
22      Mostrar “Ingrese la tercer nota: “;
23      Leer nota_3;
24      sumaParcial = sumaParcial + nota_3; // acumulo el valor de la nota 3
25      cantidadDeNotas = cantidadDeNotas + 1;  // cuento una nota más
26
27      Mostrar “El promedio es: “, sumaParcial / cantidadDeNotas;
28  FinProceso
```

En este ejemplo nuestro contador es la variable *cantidadDeNotas* y nuestro acumulador es la variable *sumaParcial*.

Vemos, como mencionamos en su definición, que el contador es una variable de tipo Entero y además, como nos sucederá en la mayoría de los casos, incrementa su valor de a 1, cada vez que me ingresan una nota.

Por otra parte vemos que la variable *sumaParcial*, nuestro acumulador es de tipo Real ya que acumula valores de tipo Real como son las notas de los alumnos. También vemos que a diferencia del contador, el acumulador va sumando valores, por lo general ingresados por el usuario, a los valores que ya tenía guardados.

Unidad Temática N°4

~ Estructuras de control ~

"Controlar la complejidad es la esencia de la programación"

-- Brian Kernigan

Estructuras de Control

Hasta ahora, en todos los ejemplos que vimos sólo podíamos asignar valores y realizar operaciones con esos valores, pero qué sucede si para mostrar el resultado (la salida) de nuestro programa debo tomar alguna decisión. Por ejemplo, en la unidad temática anterior pudimos calcular la nota promedio de una materia de un alumno, pero... ¿Si tuviésemos que imprimir por pantalla el mensaje “Materia aprobada” o “Materia desaprobada” según el promedio haya sido mayor o menor a 6? Las estructuras de control nos permiten resolver problemáticas como la planteada.

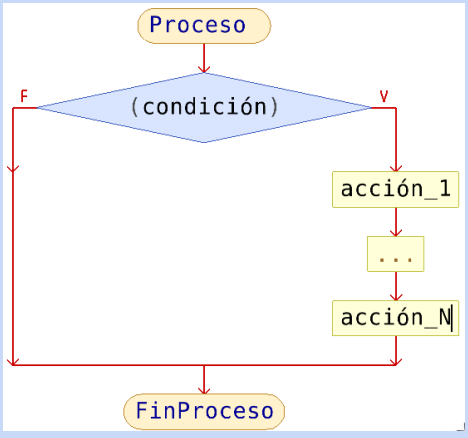
Las estructuras de control, nos permiten controlar el flujo de ejecución del programa, es decir, permiten que el programa tome distintos caminos según se cumplan ciertas condiciones. Estas condiciones deben ser *expresiones lógicas*, expresiones que arrojan un resultado Verdadero o Falso (true o false).

En el caso planteado, nuestra condición a evaluar sería *“¿El valor de promedio es mayor a 6?”*, en caso Verdadero (true) el programa debería mostrar “Materia aprobada”, en caso contrario debería mostrar “Materia desaprobada”.

Estructura de control “Si...Entonces”

Esta estructura de control posee dos alternativas. La más simple posee la siguiente forma:

Pseudocódigo	Diagrama de Flujo

Si (condición) **Entonces**
  acción_1;
  ....
  acción_N;
Fin Si;
	

¿Cómo funciona esta estructura de control?

Cuando dentro de un programa se alcanza una sentencia “Si...Entonces”, se evalúa la *condición* que se encuentra entre paréntesis. Si el resultado de esta evaluación es Verdadero (true) entonces se ejecutará la o las acciones que se encuentre dentro de este bloque. Si el resultado de evaluar la condición es Falso (false) entonces no se ejecutarán las acciones que están dentro del bloque.

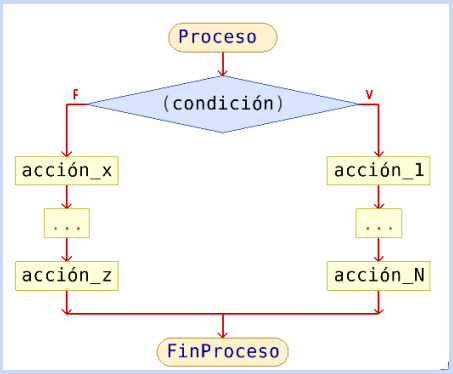
Veamos un ejemplo para clarificar:

Problema: “Ingresar un número entero. Si el número es mayor a 0 (cero) imprimir por pantalla la leyenda ‘Número mayor a 0’. Si es menor a 0 (cero) imprimir por pantalla la leyenda ‘Número menor a 0’. Si es igual a 0 (cero) imprimir por pantalla la leyenda ‘Número igual a 0’. ”

```
1  Proceso ValorIngresado
2
3      Definir numero como Entero;
4
5      Mostrar “Ingrese un número: “;
6      Leer numero;
7
8      Si ( numero > 0 ) Entonces
9          Mostrar “Número mayor a 0“;
10     Fin Si
11
12     Si ( numero < 0 ) Entonces
13         Mostrar “Número menor a 0“;
14     Fin Si
15
16     Si ( numero == 0 ) Entonces
17         Mostrar “Número igual a 0“;
18     Fin Si
19 FinProceso
```

En este ejemplo, dependiendo del número ingresado por el usuario se mostrará un mensaje u otro. Es decir, si el número es 4 sólo se mostrará el mensaje *“Número mayor a 0“*. Los otros mensaje no se mostrarán ya que 4 no es menor a 0 (cero) ni tampoco es igual a 0 (cero).

La segunda alternativa de la estructura “*Si...Entonces*”, nos permite ejecutar algunas acciones en caso de que NO se cumpla la condición evaluada. A esta alternativa la podemos llamar “Si...Entonces...Sino”. Esta alternativa tiene la siguiente forma:

Pseudocódigo	Diagrama de Flujo
Si (condición) **Entonces** acción_1; …. acción_N; **Sino** acción_x; …. acción_z; **Fin Si;**	

En este caso, cuando se alcanza una sentencia “Si…Entonces…Sino”, se evalúa la *condición* que se encuentra entre paréntesis. Si el resultado de esta evaluación es Verdadero (true) entonces se ejecutarán las acciones *acción_1…acción_N*; mientras que si el resultado de evaluar la condición es Falso (false) entonces se ejecutarán las acciones *acción_x…acción_z*.

Veamos un ejemplo para clarificar:

Problema: “Ingresar la nota de un alumno. En caso de que la nota sea mayor o igual a 6, mostrar por pantalla el mensaje ‘Materia aprobada’ . En caso contrario mostrar por pantalla el mensaje ‘Materia desaprobada’ ”

```
 1  Proceso Materia
 2
 3      Definir nota como Real;
 4
 5      Mostrar “Ingrese la nota del alumno: “;
 6      Leer nota;
 7
 8      Si ( nota >= 6 ) Entonces
 9          Mostrar “Materia aprobada “;
10      Sino
11          Mostrar “Materia desaprobada “;
12      Fin Si
13  FinProceso
```

Condiciones múltiples

Independientemente de qué forma del Si estemos utilizando siempre podemos utilizar una condición que evalúe varias cosas y no solamente un valor. Recordemos que las expresiones pueden combinarse por lo cual mientras nuestra expresión genere un valor lógico (verdadero o falso) se puede utilizar como condición en el Si.

Veamos un ejemplo con varias condiciones para clarificar:

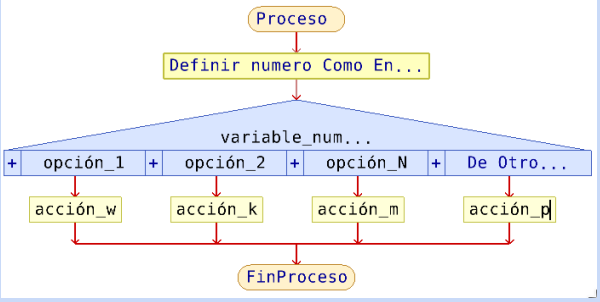
Problema: “Ingresar las 2 notas de un alumno e informar por pantalla si aprobó la materia o no. Para aprobar la materia el promedio debe ser mayor o igual a 6, y además debe haber aprobado ambos exámenes”

```
1  Proceso Materia
2
3      Definir nota1 como Real;
4      Definir nota2 como Real;
5      Definir promedio como Real;
6
7      Mostrar “Ingrese la nota del 1er examen del alumno: “;
8      Leer nota1;
9
10     Mostrar “Ingrese la nota del 2do examen del alumno: “;
11     Leer nota2;
12
13     promedio = (nota1 + nota2) / 2;
14
15     Si ( nota1 >= 6 Y nota2 >= 6 Y promedio >= 6 ) Entonces
16         Mostrar “Materia aprobada “;
17     Sino
18         Mostrar “Materia desaprobada “;
19     Fin Si
20 FinProceso
```

Estructura de control “Según”

Esta estructura de control, nos permite seleccionar una entre múltiples alternativas a ejecutar. Esta estructura es especialmente útil cuando la selección de la alternativa a ejecutar se basa en el valor de una variable numérica.

La forma para escribir esta estructura es la siguiente:

Pseudocódigo	Diagrama de Flujo
Segun variable_numérica **Hacer** opción_1: acción_w; opción_2: acción_k; ... opción_N: acción_m; **De Otro Modo:** acción_p; **Fin Segun;**	

¿Cómo funciona esta estructura de control?

Cuando dentro de un programa se alcanza una sentencia "*Segun"*, se evalúa la *variable_numérica* para saber cuál es su valor y luego se compara dicho valor con cada una de las etiquetas (opción_1, opción_2, ..., opción_N). Si alguna de esas comparaciones es Verdadera, entonces se ejecutarán las acciones de dicha etiqueta. Si el valor de la variable_numérica no coincide con alguna de las etiquetas, entonces se ejecutarán las acciones agrupadas bajo la etiqueta "De Otro Modo".

Veamos un ejemplo:

Problema: "Ingresar un número entre 1 y 3. Mostrar por pantalla el número ingresado en letras."

```
 1  Proceso Materia
 2
 3      Definir numero como Entero;
 4
 5      Mostrar "Ingrese un número: ";
 6      Leer numero;
 7
 8      Segun numero Hacer
 9          1:
10              Mostrar "uno";
11          2:
12              Mostrar "dos";
13          3:
14              Mostrar "tres";
15          De Otro Modo:
16              Mostrar "es otro número";
17      Fin Segun
18  FinProceso
```

En este ejemplo, si el usuario ingresa el número 1 o el 2 o el 3, se mostrará el mensaje “uno”, “dos” o “tres” respectivamente. Pero si el usuario ingresa el número 5, entonces se mostrará el mensaje “es otro número” ya que 5 no es una opción válida en este caso.

Anidando estructuras

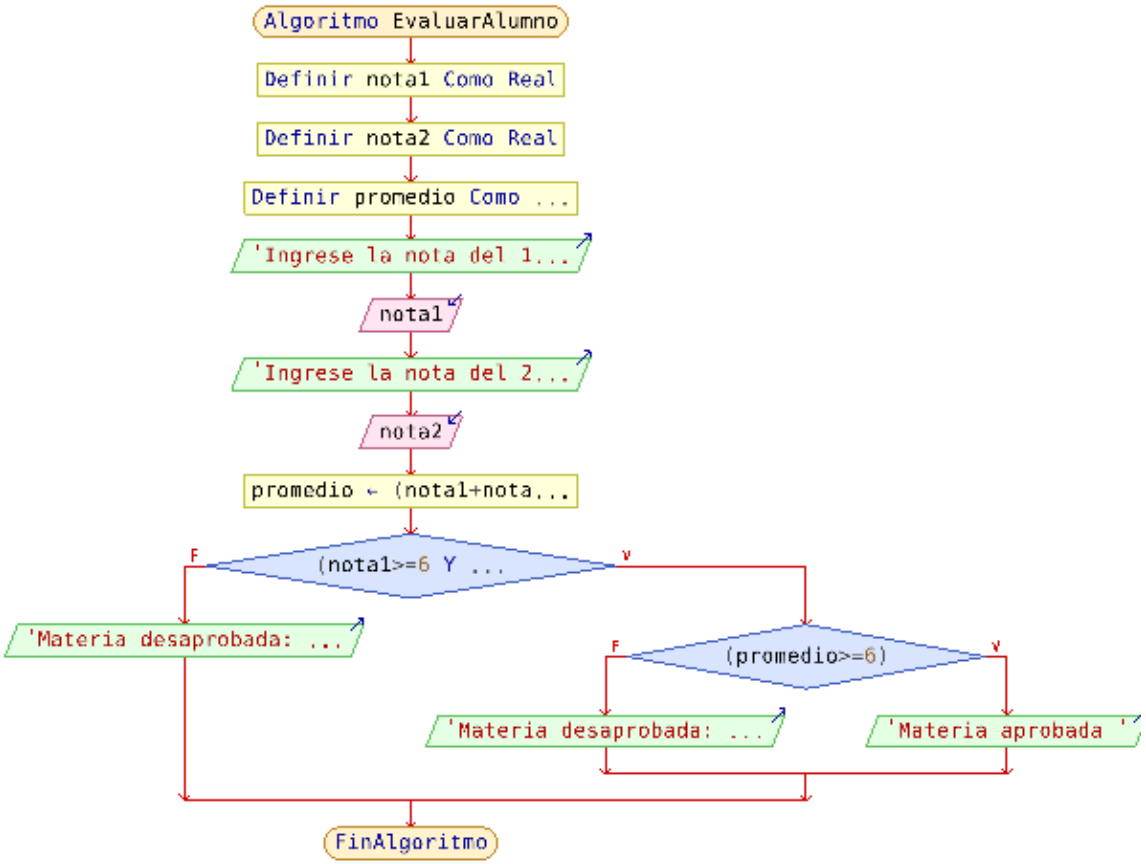
Todas las estructuras pueden combinarse y utilizarse en forma conjunta, es decir se puede realizar un Si dentro de otro Si, o un Si dentro de una de las opciones del Segun, y así infinitamente.

Es decir una estructura puede ser una de las sentencias que se ejecutan dentro de otra estructura.

Veamos algunos ejemplos:

Problema: “Ingresar las 2 notas de un alumno e informar por pantalla si aprobó la materia o no. Para aprobar la materia el promedio debe ser mayor o igual a 6, y además debe haber aprobado ambos exámenes”

```
1  Proceso Materia
2
3      Definir nota1 como Real;
4      Definir nota2 como Real;
5      Definir promedio como Real;
6
7      Mostrar “Ingrese la nota del 1er examen del alumno: “;
8      Leer nota1;
9
10     Mostrar “Ingrese la nota del 2do examen del alumno: “;
11     Leer nota2;
12
13     promedio = (nota1 + nota2) / 2;
14
15     Si ( nota1 >= 6 Y nota2 >= 6) Entonces
16         Si (promedio >= 6) Entonces
17             Mostrar “Materia aprobada “;
18         Sino
19             Mostrar “Materia desaprobada: No alcanza el promedio”;
20         Fin Si
       Sino
           Mostrar “Materia desaprobada: No aprobó ambos exámenes “;
       Fin Si
   FinProceso
```



Como se ve en el ejemplo el segundo Si se evaluará sólo si se cumple la condición del primero. De igual forma se podría poner otra estructura dentro del Sino.

Unidad Temática N°5

~ Estructuras de repetición ~

"La repetición es la madre de la habilidad".

~ Tony Robbins

Estructuras de Repetición

Hasta ahora en nuestros algoritmos cada una de las instrucciones se ejecutaba como máximo una vez.

Esto nos obliga a repetir instrucciones cuando queremos hacer más de una vez la misma operatoria.

Por ejemplo supongamos que tenemos el algoritmo que indica si una persona es mayor de edad.

```
1  Proceso Mayor
2      Definir edad como Entero;
3      Mostrar “Ingrese la edad de la persona: “;
4      Leer edad;
5      Si ( edad >= 18) Entonces
6          Mostrar “Es mayor de edad”
7      Sino
8          Mostrar “Es menor de edad“;
9      Fin Si
10 FinProceso
```

Si ahora deseamos hacer lo mismo pero para 3 personas, deberíamos triplicar todas las acciones referidas a tomar la edad y evaluarla.

```
1  Proceso Mayor
2      Definir edad como Entero;
3      Mostrar “Ingrese la edad de la persona: “;
4      Leer edad;
5      Si ( edad >= 18) Entonces
6          Mostrar “Es mayor de edad”
7      Sino
8          Mostrar “Es menor de edad“;
9      Fin Si
10
11     Mostrar “Ingrese la edad de la persona: “;
12     Leer edad;
13     Si ( edad >= 18) Entonces
14         Mostrar “Es mayor de edad”
15     Sino
16         Mostrar “Es menor de edad“;
17     Fin Si
18
19     Mostrar “Ingrese la edad de la persona: “;
20     Leer edad;
21     Si ( edad >= 18) Entonces
22         Mostrar “Es mayor de edad”
23     Sino
24         Mostrar “Es menor de edad“;
25     Fin Si
26
27 FinProceso
28
29
```

Pero...¿Y si ahora necesitamos hacerlo para 10 personas? Tendríamos que replicar las instrucciones de la línea 3 a la línea 9 **diez veces**. Lo cual hace que nuestro algoritmo sume cada vez más líneas, aunque no resuelva problemas más complejos.

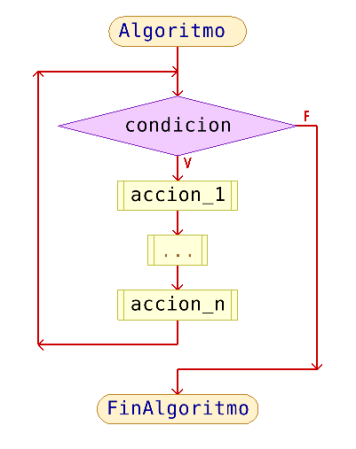
Así surge la necesidad de poder indicar en un algoritmo que ciertas instrucciones deben ejecutarse más de una vez.

Las estructuras que utilizaremos para esto las denominaremos estructuras iterativas, o estructuras de repetición, o ciclos de repetición. El objetivo de estas estructuras es generar ciclos iterativos en los que ciertas instrucciones se repitan una cierta cantidad de veces.

Estructura de repetición “Mientras...Hacer (While)”

La primer estructura iterativa que utilizaremos es el Mientras (While), esta estructura repetirá un bloque de instrucciones **mientras** se cumpla una cierta condición.

Pseudocódigo	Diagrama de Flujo	Codificación C/C++

Mientras (condición) **Hacer**
  acción_1;
  ….
  acción_N;
FinMientras
		**while** (condicion) {
		  statement_1;
		  ….
		  statement_n;
		}

¿Cómo funciona esta estructura?

Cuando dentro de un programa se alcanza una sentencia “Mientras”, se evalúa la *condición* que se encuentra entre paréntesis.

- Si el resultado de esta evaluación es Verdadero (true) entonces “entra al bucle” y se ejecutarán la o las acciones que se encuentre dentro de este bloque
- Al finalizar con estas instrucciones (terminar una iteración) se volverá a evaluar la condición.

- Cuando el resultado de evaluar la condición es Falso (false) entonces no se ejecuta la iteración.

Cabe aclarar que podría suceder que nunca se ingrese al bucle si la condición resulta Falso en la primera vez que se evalúa.

Para generar una cantidad de iteraciones, debemos pensar **qué condición** generará esa cantidad de "vueltas" dentro del ciclo de repetición. Un ejemplo es crear una variable **contador** inicializada en 0 e ir incrementando su valor dentro del ciclo.

Veamos el ejemplo anterior realizado con esta estructura:

```
Proceso Mayor
    Definir edad como Entero;
    Definir contador como Entero;
    contador = 0;

    Mientras (contador < 10) Hacer
       Mostrar "Ingrese la edad de la
       persona: ";
       Leer edad;
       Si ( edad >= 18) Entonces
          Mostrar "Es mayor de edad";
       Sino
          Mostrar "Es menor de edad";
       Fin Si
       contador = contador + 1;
    FinMientras

FinProceso
```

```
int main() {
    int edad;
    int contador;
    contador = 0;

    while (contador < 10) {
        cout << "Ingrese la edad";
        cin >> edad;
        if (edad >= 18) {
            cout << "Es mayor de edad";
        }
        else {
            cout << "Es mayor de edad";
        }
        contador = contador + 1;
    }
    return 0;
}
```

En esta solución deberemos ingresar sí o sí 10 edades, y en caso que el usuario deseara ingresar más o menos edades deberemos hacer un nuevo algoritmo.

Una forma de hacer que nuestro algoritmo sea más flexible y le permita al usuario ingresar N edades es cambiar la condición del Mientras, y en lugar de usar un valor fijo de 10 utilizar uno variable que también pueda ser ingresado por el usuario:

```
Proceso Mayor
    Definir edad como Entero;
    Definir cantidad como Entero;
    Definir contador como Entero;
    contador = 0;

    Mostrar "¿Cuántas edades ingresará?";
    Leer cantidad;

    Mientras (contador < cantidad) Hacer
```

```
int main() {
    int edad;
    int contador;
    int cantidad;
    contador = 0;

    cout <<  "¿Cuántas edades ingresará?";
    cin >> cantidad;
    while (contador < cantidad) {
        cout << "Ingrese la edad";
```

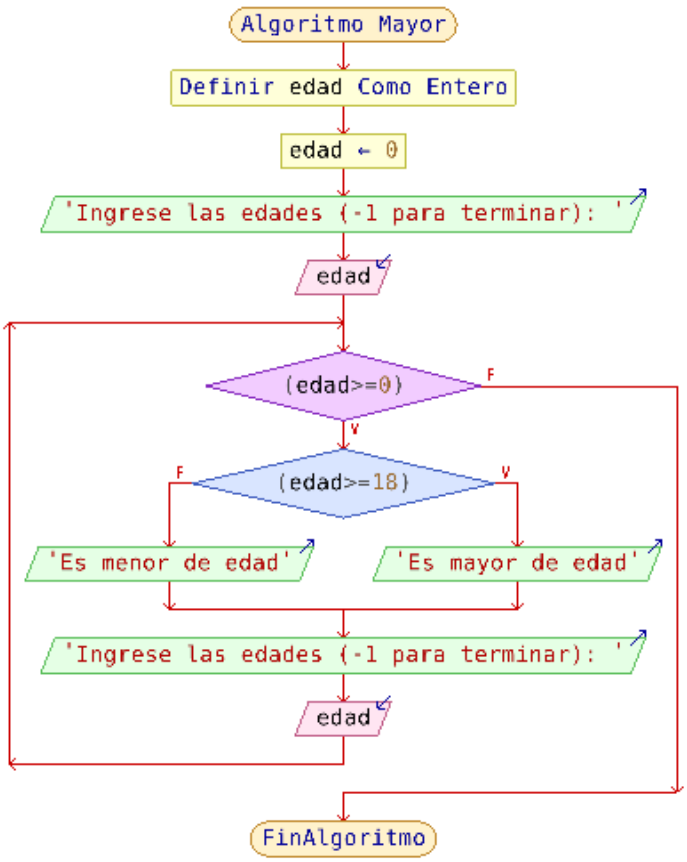
```
11        Mostrar “Ingrese la edad de la
12        persona: “;
13        Leer edad;
14        Si ( edad >= 18) Entonces
15           Mostrar “Es mayor de edad”;
16        Sino
17           Mostrar “Es menor de edad“;
18        Fin Si
19        contador = contador + 1;
20     FinMientras
21
22  FinProceso
```

```
        cin >> edad;
        if (edad >= 18) {
            cout << “Es mayor de edad”;
        }
        else {
            cout << “Es mayor de edad”;
        }
        contador = contador + 1;
    }
    return 0;
}
```

Ahora nuestro algoritmo permitirá ingresar N cantidad de edades.

Otra forma de resolverlo hubiese sido que el usuario ingresara un valor negativo para indicar que quiere terminar la carga (dado que no hay edades negativas).

El diagrama de esta solución sería:



A diferencia de la solución anterior, aquí no tenemos necesidad de contar ya que no estamos buscando generar una cantidad de iteraciones conocidas, sino que el ciclo se cortará al ingresar un valor negativo.

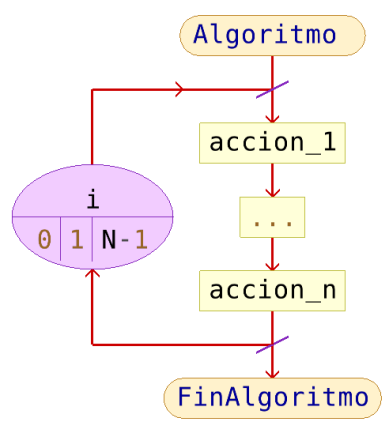
Algo a tener en cuenta es que debo tomar el primer valor fuera del ciclo para no terminar utilizando valores de corte en las instrucciones. En lugar de incrementar un contador como últimas instrucciones dentro del ciclo, lo que tendremos es el nuevo ingreso de una edad.

Esta estructura es de las que llamamos **No exactas** porque la cantidad de iteraciones podría variar en la ejecución.

Estructura de repetición “Para...Hacer (For)”

La estructura Para funciona de manera similar al Mientras pero establece dentro de la misma estructura una variable que contará las iteraciones. Por esta razón decimos que el Para es una estructura de iteración de repeticiones **Exactas.**

Pseudocódigo	Diagrama de Flujo	Codificación C/C++

Para i<-0 **Hasta** N-1 **Hacer**
  accion_1
  ...
  accion_n
FinPara
		**for** (int i = 0; i < N; i++) {
		  statement_1;
		
		  ....
		
		  statement_n;
		}

¿Cómo funciona esta estructura?

En el ciclo Para se utiliza lo que se llama la **variable del índice**. Esta variable tiene que estar definida como una variable de tipo Entero. En nuestro ejemplo, la variable de control es **i**. La variable de control se inicializa con un número (en este caso 0) e irá tomando todos los valores enteros hasta llegar al número **N -1**

¿Cómo se realiza esta asignación?

Luego de inicializarse (en su valor inicial), se evalúa si ya se alcanzó el valor final, en caso que aún no lo haya alcanzado se ejecuta el cuerpo del ciclo que se compone de 1 o más instrucciones (bloque).

Luego de ejecutar el cuerpo del ciclo, se incrementa en uno el valor de **i**, es decir, que internamente se realiza una asignación **i = i + 1**

Luego, el ciclo "Para" evalúa el valor de **i** y si todavía no llegó a **N - 1** (el valor final) vuelve a ejecutar el cuerpo del ciclo. Y así sucesivamente hasta que **i** alcanza su valor final.

Es importante destacar que el incremento de la variable del índice no se debe realizar en el algoritmo, ya que forma parte del funcionamiento de la estructura del "Para". Esta es una diferencia con respecto a los otros ciclos en donde debemos asegurarnos que la condición se modifique para poder finalizar.

También es importante que comprendas que los valores iniciales y finales son también válidos, es decir que la primera iteración se ejecutará con **i = 0** , y la última iteración se ejecutará con **i = N - 1**

Como el Para es un ciclo de repeticiones exactas, necesitaré saber de antemano cuántas iteraciones quiero generar, ya sea porque el usuario lo ingresó, o porque es una cantidad conocida

Rehagamos el ejemplo anterior con un ciclo Para

```
Proceso Mayor
    Definir edad como Entero;
    Definir cantidad como Entero;

    Mostrar "¿Cuántas edades ingresará?";
    Leer cantidad;
    Para i<-0 Hasta cantidad - 1 Hacer
       Mostrar "Ingrese una edad: ";
       Leer edad;
       Si ( edad >= 18) Entonces
          Mostrar "Es mayor de edad";
       Sino
          Mostrar "Es menor de edad";
       Fin Si
    FinPara

FinProceso
```

```
int main() {
    int edad;
    int contador;

    cout <<  "¿Cuántas edades ingresará?";
    cin >> cantidad;
    for (int i = 0; i < cantidad; i++) {
        cout << "Ingrese una edad";
        cin >> edad;
        if (edad >= 18) {
            cout << "Es mayor de edad";
        }
        else {
            cout << "Es mayor de edad";
        }
    }
    return 0;
}
```

Veamos paso a paso cómo se ejecuta el algoritmo:

Paso	cantidad	edad	i	Mensaje	Comentario
5	-	-	-	-	Se imprime el mensaje solicitando una cantidad de edades
6	**3**	-	-	-	Se lee la cantidad ingresada
7	3	-	0	-	Se define e inicializa la variable i Se controla que se cumpla la condición del Para
8	3	-	0	-	Se imprime el mensaje pidiendo una edad
9	3	**23**	0	-	Se lee la edad ingresada
10	3	23	0	-	Evaluar la condición edad >= 18
11	3	23	0	**“Es mayor”**	Por ser verdadera la evaluación se imprime el mensaje “Es mayor de edad”
12	3	23	**1**	-	Finaliza la primer iteración Incrementa el contador en 1
13	3	23	1	-	Vuelve a evaluar la condición del Para(i < cantidad) Como se cumple, sigue en el bucle y comienza la 2da iteración
8’	3	23	1	-	Se imprime el mensaje pidiendo una edad
9’	3	**12**	1	-	Se lee la edad ingresada
10’	3	12	1	-	Evaluar la condición edad >= 18
11’	3	12	1	**“Es menor”**	Por ser falsa la evaluación se imprime el mensaje “Es menor de edad”
12’	3	12	**2**	-	Finaliza la segunda iteración Incrementa el contador en 1
13’	3	12	2	-	Vuelve a evaluar la condición del Para(i < cantidad) Como se cumple, sigue en el bucle y comienza la 3era iteración
8’’	3	12	2	-	Se imprime el mensaje pidiendo una edad
9’’	3	**54**	2	-	Se lee la edad ingresada
10’’	3	54	2	-	Evaluar la condición edad >= 18
11’’	3	54	2	**“Es mayor”**	Por ser verdadera la evaluación se imprime el mensaje “Es mayor de edad”
12’’	3	54	**3**	-	Finaliza la tercer iteración Incrementa el contador en 1
13’’	3	54	**3**	-	Vuelve a evaluar la condición del Para(i < cantidad) Como no se cumple, porque ambos valen 3 sale del bucle
14	3	54	**-**	-	Finaliza la ejecución (retorna 0)

Cambiando el incremento

Como se detalla en la descripción paso a paso de cómo se modifica el contador del **Para**, al finalizar cada iteración se incrementa el valor de la variable índice (la i en nuestro caso).

En los ejemplos que vimos el incremento siempre fue de 1 es decir que i toma el valor i + 1, o de otra forma i = i + 1 (que es lo mismo que i++ en c). Este valor en que se incrementa la variable contador es llamado **paso** o **incremento** del Para, y puede modificarse.

Veamos un ejemplo, si desearamos generar los números pares 2, 4, 6, 8, 10 y 12 utilizando un ciclo de repetición podríamos generar todos los números desde 2 al 12 (el 0 y 1 los salteamos porque no nos interesan) y por cada número preguntamos si es par o no.

```
1  Proceso Pares
2      Para i<-2 Hasta 12 Hacer
3          Si ( i % 2 == 0) Entonces
4             Mostrar “Es par:”, i;
5          Fin Si
6      FinPara
7  FinProceso
8
```

```
int main() {
    for (int i = 2; i <= 12; i++) {
        if (i % 2 == 0) {
            cout << “Es par:” << i;
        }
    }
    return 0;
}
```

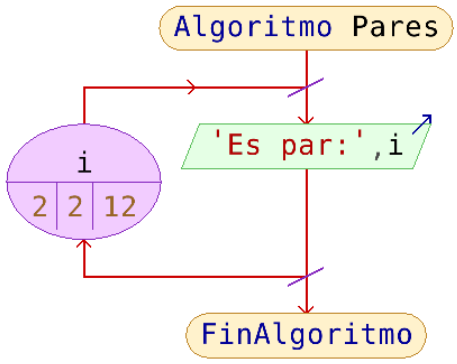
La anterior es una solución completamente válida, y generará los números deseados. Pero modificando el paso o incremento existe una solución mucho más sencilla en la que no es necesario preguntar si el valor del contador es par o no. Dado que sabemos que si el contador arranca en 2, si el paso fuera de 2 también entonces todos los números generados serían pares, por lo tanto no es necesario el condicional y podemos eliminarlo.

```
1  Proceso Pares
2      Para i<-2 Hasta 12 Con Paso 2 Hacer
3          Mostrar “Es par:”, i;
4      FinPara
5  FinProceso
6
```

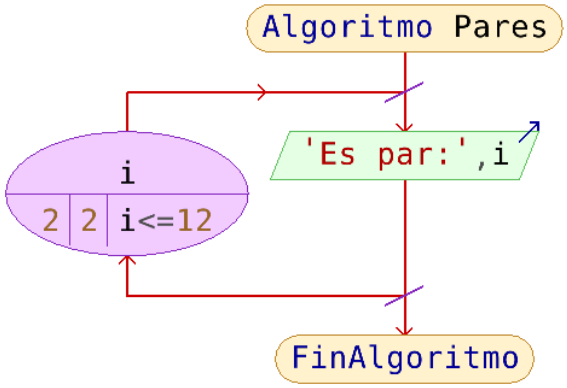
```
int main() {
    for (int i = 2; i <= 12; i = i + 2) {
        cout << “Es par:” << i;
    }
    return 0;
}
```

En el diagrama aunque no lo hayamos aclarado anteriormente, existe un lugar específico donde se indica el incremento o paso, y es la posición del medio de la burbuja. Anteriormente no le prestamos atención pero si se mira el diagrama del ejemplo se verá que en el medio se encuentra el valor 1, ya que estábamos usando incrementos de 1.

Veamos cómo quedaría el diagrama del algoritmo de Pares mejorado:



Si quisiéramos usar la misma condición que en el código c/c++ el diagrama quedaría de la siguiente forma:



En lugar de el valor final, colocamos la condición de corte. Ambos diagramas son equivalentes. Recuerde que si hay un valor final, este es **inclusive**

Anidando estructuras de repetición

Al igual que con las estructuras de control, las estructuras de repetición pueden contener entre sus acciones otras estructuras.

Si revisamos algunos de los ejemplos vistos hasta ahora veremos que dentro de las acciones a realizar en cada iteración había una estructura condicional, recordemos siempre que las estructuras establecen bloques de código y estos pueden anidarse indefinidamente en caso de ser necesario.

Lo particular de anidar estructuras de repetición es que generará una cantidad iteraciones de la estructura anidada por cada iteración de la estructura de repetición principal.

Veámoslo con un ejemplo, supongamos que deseamos calcular el promedio de notas de todo el curso. Sabiendo que tenemos 30 alumnos y 4 notas por alumno, el algoritmo resultaría:

```
Proceso Promedios
    Definir nota como Entero;
    Definir prom como Real;
    Definir nombre como Cadena
    Para i<-0 Hasta 29 Hacer
        Mostrar “Ingrese nombre”;
        Leer nombre;
        prom = 0;     // Inicializa prom
        Para j<-0 Hasta 3 Hacer
           Mostrar “Ingrese nota”;
           Leer nota;
           prom = prom + nota;  //acumula
        FinPara
        prom = prom / 4;  //Calcua el prom
        Mostrar “Promedio:”, prom;
    FinPara
FinProceso
```

```
int main() {
    int nota;
    float prom;
    string nombre;
    for (int i = 0; i < 30; i++) {
        cout << “Ingrese nombre:”;
        cin >> nombre;
        prom = 0;
        for (int j = 0; j < 4; j++) {
            cout << “Ingrese una nota”;
            cin >> nota;
            prom = prom + nota;
        }
        prom = prom / 4;
        cout << “Promedio:” << prom;
    }
    return 0;
}
```

El primer Para, generará 30 iteraciones (una para cada alumno) dado que el contador i tomará los valores 0, 1, 2, … , 29. Mientras que el segundo Para genera 4 iteraciones (una por cada nota del alumno) dado que el contado j tomará los valores 0, 1, 2, 3.

Al estar el Para de las notas anidado en el el Para de los alumnos resulta que para cada iteración de alumno, se generan 4 iteraciones de notas, es decir, por cada alumno se solicita el ingreso de 4 notas. Que es lo que queríamos lograr.

El anidamiento de estructuras no está limitado a los Para obviamente, sino que se pueden anidar ciclos Mientras, o combinar un ciclo Mientras con uno de tipo Para o viceversa, **todo dependerá del problema a solucionar**.

Veamos un ejemplo donde se combina un Mientras con un Para: Supongamos que, siguiendo el mismo ejemplo anterior, el curso no necesariamente es de 30 alumnos sino que el Profesor al realizar la carga decide terminarla ingresando el nombre “Fin”.

Recordemos que por no saber a priori cuántos alumnos se desean cargar no podremos usar un ciclo exacto como el Para, sino que tendremos que utilizar un Mientras para generar las iteraciones de cada alumno, sin embargo si sabemos que son 4 notas por cada alumno por lo cual podríamos usar un Para como estructura de repetición para pedir las notas.

```
Proceso Promedios
    Definir nota como Entero;
    Definir prom como Real;
    Definir nombre como Cadena;
    Mostrar “Ingrese nombre”;
    Leer nombre;
    Mientras nombre != “Fin” Hacer
        prom = 0;     // Inicializa prom
        Para j<-0 Hasta 3 Hacer
           Mostrar “Ingrese nota”;
           Leer nota;
           prom = prom + nota;  //acumula
        FinPara
        prom = prom / 4;  //Calcua el prom
        Mostrar “Promedio:”, prom;

        Mostrar “Ingrese nombre”;
        Leer nombre;
    FinMientras
FinProceso
```

```
int main() {
    int nota;
    float prom;
    string nombre;
    cout << “Ingrese nombre:”;
    cin >> nombre;
    while (nombre != “Fin”) {
        prom = 0;
        for (int j = 0; j < 4; j++) {
            cout << “Ingrese una nota”;
            cin >> nota;
            prom = prom + nota;
        }
        prom = prom / 4;
        cout << “Promedio:” << prom << endl;

        cout << “Ingrese nombre:”;
        cin >> nombre;
    }
    return 0;
}
```

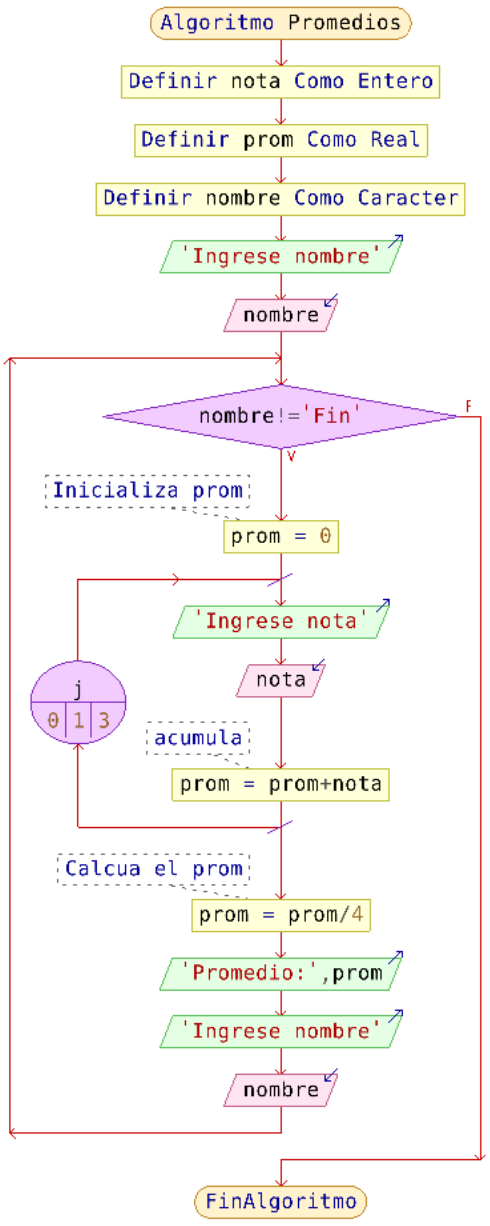
En los ejemplos que hemos visto del Mientras la condición de corte siempre estaba vinculada a una variable contador que se actualizaba dentro del ciclo. Sin embargo en este ejemplo vemos que la condición no es un contador ya que no sabemos qué cantidad de iteraciones debemos generar, en este caso la condición de corte evalúa el valor ingresado por el usuario.

De igual forma que antes se inicializaba el contador en 0 antes del Mientras, ahora debemos tomar un valor ingresado por el usuario antes del Mientas para poder evaluarlo. Luego las últimas acciones de cada iteración serán volver a pedir el valor.

El resultado de este algoritmo es que se pedirán nuevos alumnos y sus 4 notas mientras que nose ingrese como nombre “Fin”, con lo cual tenemos un ciclo inexacto para los alumnos (ya no

sabemos con seguridad cuántos alumnos se ingresarán) y un ciclo exacto anidado que permite ingresar las 4 notas del alumno (esto sigue como en el ejemplo anterior).

Veamos el diagrama del algoritmo anterior:



TODO: Otros Bucles menos usados: Hacer...Mientras y Hacer...Hasta

Unidad Temática N°6

~ Funciones y Procedimientos~

“La primera regla de las funciones es que deben ser pequeñas. La segunda regla de las funciones es que deberían ser más pequeñas que eso ".

~ Robert C. Martin

Funciones y Procedimientos

En esta unidad explicaremos el concepto de división por módulos, sus principales características y las razones para utilizarlo. También veremos cómo dividir en módulos un programa utilizando funciones y procedimientos.

En las unidades anteriores vimos las estructuras de control para la toma de decisiones. También los ciclos iterativos con los cuales se pueden realizar un conjunto de instrucciones más de una vez.

A lo largo de las actividades, los algoritmos se van haciendo cada vez más complejos y con una mayor cantidad de acciones. Es por eso que necesitamos una estrategia para poder construir programas más complejos.

Utilizaremos la división por módulos para dividir el problema en subproblemas y, de esa manera, simplificar el desarrollo.

Supongamos un problema de la vida real.

Imagínate que decides mudarte, no importa el motivo, piensa solamente que te vas a mudar. Ahora bien, para resolver este “problema” necesitas hacer varias tareas, por ejemplo, buscar un nuevo hogar, adquirir la propiedad y, finalmente, realizar la mudanza.

Esto que acabamos de realizar fue dividir un problema muy grande en tres problemas más pequeños. Al resolver estos tres problemas, estarás resolviendo el problema “grande” que los abarca.

A su vez el problema de **buscar un nuevo hogar**, se puede dividir, en otros subproblemas como pueden ser:

- visitar inmobiliarias,
- navegar sitios web,
- comparar propiedades,
- visitar las propiedades,
- decidir cuál adquirir.

Como habrás notado, estamos dividiendo cada problema en problemas menores.

Ahora, ¿por qué hacemos esto? Principalmente porque es mucho más fácil resolver un problema pequeño que uno mayor.

Desarrollo Top Down

La metodología de división por módulos se conoce habitualmente como “divide y vencerás” y en programación se llama **Desarrollo Top Down**.

Pensar en el problema general e ir descomponiéndolo en subproblemas. A su vez, estos subproblemas se podrán seguir dividiendo hasta llegar a un subproblema lo bastante simple como para poder resolverse de forma sencilla.

Así es como realizaremos un algoritmo principal y varios subalgoritmos para ir resolviendo cada uno de los subproblemas.

Estos subalgoritmos, en programación, tienen distintos nombres, pero representan el mismo concepto. Es por eso que podemos encontrarnos con los nombres de: módulos, subprogramas, rutinas, subrutinas. Pero son todos sinónimos para definir el mismo concepto de “subproblema”.

Razones para Modularizar

Ya vimos que una de las razones para definir módulos es que resolver un problema más simple es mucho más fácil que resolver un problema complejo. Ahora bien, hay otros beneficios al modularizar un algoritmo.

Simplifica el programa: Un módulo es más simple de programar.

Permite la reutilización de código: La reutilización de código es una característica fundamental de la programación. Una vez que se programa un módulo, si ese mismo módulo hace falta para resolver otro problema, se puede utilizar sin necesidad de pensar nuevamente cómo resolverlo. Por ejemplo, si realizaste un módulo para calcular una raíz cuadrada, ese módulo lo podrás

reutilizar en cualquier algoritmo que tenga que calcular una raíz cuadrada. Es un subproblema que ya resolviste en otro momento y no tenés que pensar nuevamente en él, porque ya sabes que está resuelto. De esta manera evitamos lo que llamamos comúnmente “reinventar la rueda”.

El algoritmo resulta más simple y más claro: En lugar de tener una lista muy larga de instrucciones que puede llegar a ser ilegible, el algoritmo principal será una pequeña lista de módulos. De esta manera, al leer el algoritmo principal, se entenderá la idea de lo que realiza, sin necesidad de acceder a los detalles de cómo lo resuelve.

El programa es más simple de verificar: Las pruebas se pueden realizar en cada módulo de forma independiente, en lugar de tener que probar todo el programa a la vez.

Reduce los tiempos de mantenimiento: Realizar modificaciones en un programa es más simple y rápido porque dividir en módulos aísla los cambios. Una vez que se detecta en dónde realizar el cambio se deberá modificar ese módulo solamente. Sabiendo que el resto del Algoritmo debería funcionar como lo hacía anteriormente.

Abstracción

Veremos ahora un concepto fundamental en el desarrollo de programas que se deriva de la técnica de división por módulos. Este concepto es la Abstracción.

Concretamente, la abstracción se produce cuando creamos módulos.

Por ejemplo; si tenemos un algoritmo principal que se encuentra dividido en tres módulos de la siguiente manera:

```
Algoritmo Principal
    Obtener Notas de Un alumno
    Calcular Promedio De Notas
    Mostrar Promedio de Notas
Fin AlgoritmoPrincipal
```

Acá estamos aprovechando una de las ventajas de la división en módulos. Como comentamos previamente, el algoritmo es más claro ya que, con los nombres que hemos dado a los módulos, queda bien especificado qué es lo que hace cada uno de ellos.

Analicemos qué sucede con el Módulo “Calcular Promedio De Notas” ...

Seguramente estarás pensando que este módulo calcula el promedio de unas notas, pero no se ocupará ni de pedir las notas ni de mostrar los resultados. Este módulo está **abstraído del resto**, solo se ocupará de calcular el promedio. No sabemos tampoco cómo lo resolverá, ya que de eso nos ocuparemos cuando pasemos a desarrollar este módulo.

Es la idea de una **“caja negra”**, sabemos qué necesita y qué hace el módulo, pero no sabemos cómo lo realiza.

Los módulos son independientes entre sí, aunque algunos pueden necesitar colaborar con otros, o trabajar de forma conjunta. Veremos, en esta misma unidad, cómo lograr que los módulos se puedan “comunicar” o transmitir datos entre ellos a través de los parámetros.

Lo importante en este momento, para entender el concepto de abstracción, es comprender que **cada módulo es independiente de los demás módulos y que es ideal que realice una sola tarea**.

Podemos definir la abstracción como el **aislamiento de un elemento de su contexto o del resto de los elementos que lo acompañan.**

Ocultamiento de la información

Un concepto que se deriva de la abstracción es el de Ocultamiento de la información.

Continuamos con el ejemplo "Calcular Promedio de Notas"....

Para calcular el promedio, seguramente se realizarán operaciones intermedias, por ejemplo, sumar todos los números, contar la cantidad de números, etc. Esa información forma parte de la manera en que el módulo va a resolver su problema, no es de interés para el resto de los módulos, es por eso que esa información queda “oculta” para el resto del sistema.

Comunicación entre Módulos

Si analizamos el ejemplo anterior, entendemos que cada módulo tiene una única tarea, y que nos oculta la forma en que la va a realizar, pero...¿Puede hacer lo que necesita sin interactuar con nadie más?

Pensémoslo un poco más ¿Es posible que el módulo *Calcular Promedio de Notas* realice el promedio sin recibir ninguna información? Y yendo un poco más allá. ¿De qué forma el Algoritmo principal se entera del resultado del cálculo?

Se tiene que definir alguna forma en la cual cada módulo pueda recibir la información necesaria para realizar su tarea, y al mismo tiempo otro mecanismo para que informe el resultado obtenido a quién lo necesite. Para esto es que existen los Parámetros y Resultado (o retorno) del Módulo.

Por ejemplo, el módulo *Calcular Promedio de Notas* necesitará recibir las notas a promediar, para lo cual definirá los parámetros de entrada, y también definirá que al finalizar su ejecución retornará un valor real que será el promedio calculado.

Si se analiza pensando los pasos del análisis del problema de las primeras unidades, vemos que cada módulo puede pensarse como un algoritmo en sí mismo, esto es fruto de la abstracción y es lo que permite achicar los problemas hasta que sean más fáciles de resolver.

Tipos de Módulos

Existen dos tipos de módulos, según su declaración y utilización. Las funciones y los procedimientos.

Función

El concepto “función” no es nuevo ya que lo hemos usado en matemática. Una función matemática tiene parámetros que pertenecen al Dominio de esa función y las funciones devuelven un resultado de acuerdo al valor de dichos parámetros.

En programación los **módulos** que llamamos Funciones tienen el mismo comportamiento, de allí su nombre. En general son módulos que realizan alguna operación sobre sus parámetros o, como lo llamaremos, datos de entrada, y **devuelven un único resultado**.

Funciones en C/C++

La forma genérica de una función en C/C++ es la siguiente

```
tipo_de_retorno nombre_funcion( lista_parametros ) {
  cuerpo de la funcion
}
```

Consta de una definición o encabezado (también llamado header) y un conjunto de instrucciones (también llamado body o cuerpo). Veamos la definición de de cada parte:

- **Tipo de Retorno:** Las funciones retornan un resultado. El tipo de retorno es el tipo de dato del valor que la función retorna.
- **Nombre de Función:** Es el nombre de la función. El nombre y la lista de parámetros constituyen lo que llamamos firma de la función.
- **Parámetros:** Un parámetro es un contenedor de valor. Cuando una función es llamada se pasa un valor para cada parámetro declarado, este valor es referido como argumento. La lista de parámetros se refiere al tipo, orden y número de parámetros de una función. Los parámetros son opcionales; es decir una función puede no tener parámetros.
- **Cuerpo de la Función:** El cuerpo de la función contiene el conjunto de sentencias que define lo qué se ejecuta al invocar la función.

Si revisamos los ejemplos de código de unidades anteriores veremos que en realidad ya estábamos usando una función, que es la función principal o main. Recordemos el ejemplo del Para de la unidad anterior

```
1  int main() {
2      for (int i = 2; i <= 12; i++) {
3          if (i % 2 == 0) {
4              cout << “Es par:” << i;
5          }
6      }
7      return 0;
8  }
```

Cómo podemos ver, la función main no recibe parámetros de entrada y tiene un tipo de retorno int, por lo cual al final siempre tenemos un return.

Cabe destacar que si la función define un tipo de retorno entonces el algoritmo de la función SIEMPRE debe retornar un valor de ese tipo.

Veamos otro ejemplo, la siguiente función devuelve el valor máximo de los dos valores recibidos por parámetro.

```
1   // funcion que retorna el máximo de dos valores
2
3   int max(int num1, int num2) {
4      // declaracion de variable local
5      int result;
6
7      if (num1 > num2)
8          result = num1;
9      else
10         result = num2;
11
12     return result;
13  }
```

En este caso si recibimos parámetros que son los dos números a comprar, y retornaremos el valor que tengamos almacenado en result.

La variable result, está definida a nivel local, esto quiere decir que la variable solo se puede utilizar dentro de esta función, y al retornar la variable en realidad lo que estamos retornando es solo el valor actualmente almacenado en la misma.

Es decir, **las variables locales sólo pueden ser accedidas, leídas y modificadas por el módulo que las declara**. Los otros módulos no conocen su existencia ni pueden consultarlas.

Pero entonces, ¿cómo es que alguien podría obtener el valor que devuelve la función? Justamente a través de la invocación de la función con los parámetros necesarios. El resultado de una función puede ser utilizado en expresiones, asignaciones, etc.

El objetivo de una función es “extender” las instrucciones primitivas con las que ya cuenta el lenguaje.

Invocando funciones

Mientras que al definir o declarar una función, se debe indicar qué hace (las instrucciones que la definen). Para usar una función deberás llamarla o invocarla.

Cuando en un programa se llama a una función, el control del programa o el hilo de ejecución es transferido a la función llamada. La función llamada realizará las tareas definidas y al llegar a la instrucción return o al llegar a la llave que finaliza la función en caso de no tener retorno (luego veremos que estos módulos son llamados “procedimientos”), retorna el control al programa principal.

Para llamar una función simplemente se debe pasar los parámetros requeridos junto con el nombre de la función, y si la función retornara un valor, se puede almacenar en una simple asignación. Por ejemplo

```
    Ejemplo declarando toda la función antes del main.

1   #include <iostream>
2   using namespace std;
3
4   // decalracion completa de la funcion
5   int max(int num1, int num2) {
6      int result;
7
8      if (num1 > num2)
9          result = num1;
10     else
11         result = num2;
12     return result;
13  }
14
15  int main () {
16     // variables locales:
17     int a = 100;
18     int b = 200;
19     int ret;
20
21     // llamado al a funcion.
22     ret = max(a, b);
23     cout << "Maximo: " << ret << endl;
24
25     return 0;
26  }
27
```

```
Ejemplo declarando la firma antes del main y cuerpo luego.

#include <iostream>
using namespace std;

// decalracion de la firma de la funcion
int max(int num1, int num2);

int main () {
   // local variable declaration:
   int a = 100;
   int b = 200;
   int ret;

   // llamado a la funcion.
   ret = max(a, b);
   cout << "Maximo: " << ret << endl;

   return 0;
}

// declaracion de la funcion
int max(int num1, int num2) {

   if (num1 > num2)
       return num1;
   else
       return num2;
}
```

Como se puede ver en los ejemplos, la declaración de la función debe estar antes del main para poder utilizarla. En el caso de la izquierda se declara la función completa, mientras que en el caso de la derecha solamente se declara la firma, y la definición completa al final.

Declarar las firmas de las funciones y no las funciones completas es algo que permite separar la definición de la función de la implementación y permitirá más adelante separar en archivos las declaraciones en archivos header o .h y las implementaciones en archivos .cpp. Esto permite tener el más visible la función main y no tener que ir hasta muchas líneas de código más abajo.

En el cuerpo de la función de la derecha también se muestra otra forma que podría definir la búsqueda del mayor, sin necesidad de utilizar otra variable local sino retornando directamente los valores recibidos.

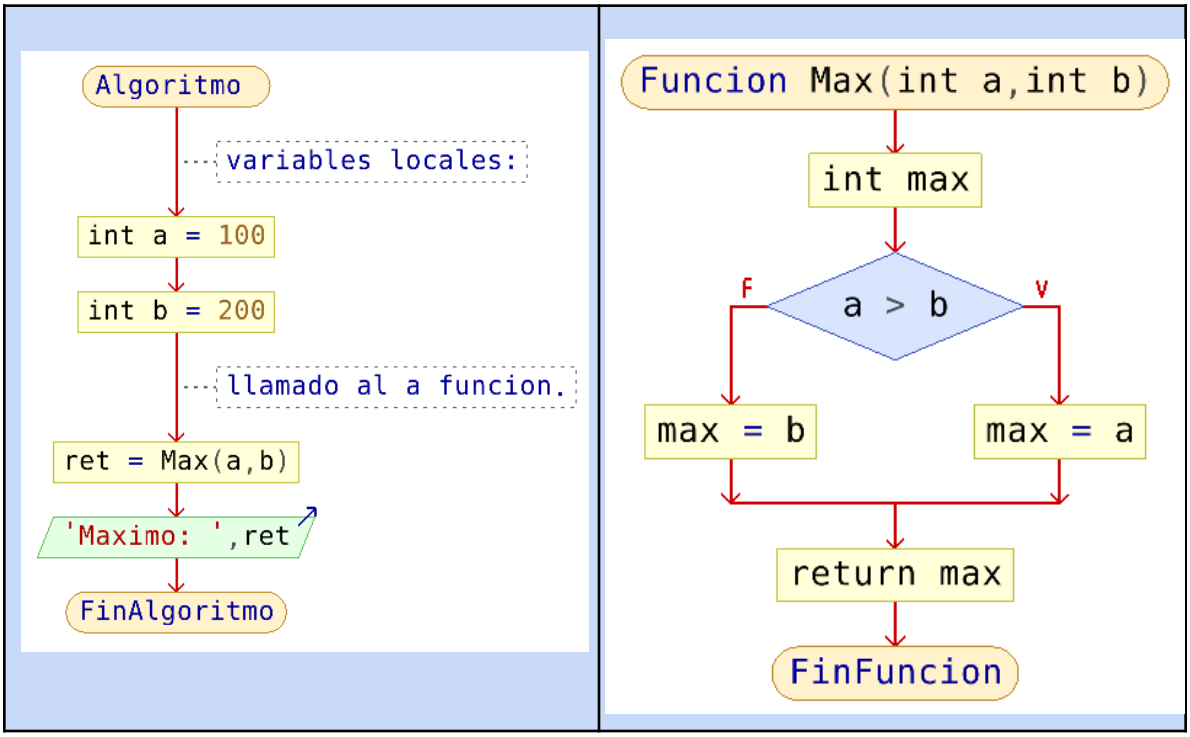
Cabe destacar que en el programa principal las variables locales son a y b, y al llamar a la función estas son las variables utilizadas, entonces a y b serán lo que llamamos “argumentos”, es decir la función en sus parámetros num1 y num2 recibirá los valores de a y b respectivamente. Vale la insistencia, lo que se pasa como parámetro en este caso es solo el valor y no la variable. Esto es lo que permite el ocultamiento y protección de la información: La función main no conoce

ni puede acceder a las variables locales de max, y viceversa. Toda la comunicación se realiza a través de los parámetros recibidos.

Diagramas

Para realizar los diagramas, lo que se hará es un diagrama independiente para cada módulo, y en el algoritmo principal (main) se mostrarán las invocaciones como si fueran expresiones.

Veamos el diagrama del algoritmo anterior:



Ámbito de las variables

Como vimos en la Unidad 2, una **variable es un espacio en memoria asociado a un nombre lógico**.

La **memoria total** que se encuentra disponible en un equipo informático está dividida en distintas áreas o sectores destinados a diferentes componentes del sistema. Por ejemplo, habrá un área de memoria para el sistema operativo, otra área de memoria para administrar las aplicaciones, y también habrá un área de memoria específica dedicada exclusivamente para nuestro programa.

Es por eso que las variables y todos los componentes de nuestro algoritmo utilizarán ese espacio de memoria. Estas secciones son conocidas como ámbitos de ejecución.

Las **variables, constantes, módulos y todo el algoritmo** serán almacenados en el ámbito del programa. Pero del mismo modo que te explicamos el concepto de dividir por módulos un

algoritmo, el área de memoria se encuentra, también, dividida generando los distintos ámbitos de memoria de cada rutina.

Por lo tanto, tendremos un ámbito de memoria global para nuestro algoritmo principal, y cada rutina tendrá definido un ámbito de memoria local, el cual será de acceso exclusivo dentro de esa rutina.

La división de estos ámbitos nos permite **clasificar las variables en globales y locales**.

Variables Globales

Las Variables Globales son aquellas que se definen en el programa principal (afuera del main). Si bien estas variables están accesibles desde el algoritmo principal y desde todos los módulos, es un error grave acceder directamente a ellas desde un módulo.

Para acceder a una variable global desde un módulo se debe pasar la variable como un parámetro. De esta manera logramos mantener la abstracción, ya que desde el módulo no se tendrá acceso directamente a un componente que no se encuentra definido en él.

A las Variables Globales se debe acceder solamente desde el programa principal, de lo contrario no estaremos realizando una abstracción de los módulos.

Declaración de variables globales en C / C++

```
1     #include <iostream>
2     using namespace std;
3
4     // Declaracion de variable global:
5     int g;
6
7     int main () {
8        // Decalracion de variable local:
9        int a, b;
10
11       // inicializacion
12       a = 10;
13       b = 20;
14       g = a + b;
15       cout << g;
16       return 0;
17    }
```

Un programa podría tener una variable local y otra global con el mismo nombre, pero la variable declarada en forma local tendrá preferencia y será la utilizada. Ejemplo

```
1     #include <iostream>
2     using namespace std;
3
4     // Variable global:
5     int g = 20;
6
7     int main () {
8        // Variable local:
9        int g = 10;
10
11       cout << g;
12
13       return 0;
14    }
```

Cuando el código anterior se compile y ejecute, se produce el siguiente resultado:

```
10
```

Variables Locales

Las variables locales son aquellas que se definen dentro de un módulo. A estas variables se pueden acceder solamente dentro del mismo módulo, ya que se encuentran almacenadas en su ámbito. Cuando un módulo se invoca, se crea en memoria el ámbito de ese módulo, que es donde se almacenan los parámetros y las variables locales.

Una vez que finaliza, se elimina su ámbito de memoria, borrándose todos los parámetros y variables locales.

A las Variables Locales solo se puede acceder desde el módulo que las define

En C / C++ existen dos formas de declarar variables que llamamos locales:

- Dentro de un bloque de Código o de una función.
- En la definición de una función como parámetros.

Retomando el ejemplo de la función max, vemos que la función tendrá las siguientes variables locales:

- Definidas por parámetros: **num1** y **num2**
- Definidas localmente: **result**

Mientras que la función main tiene las siguientes variables locales:

- Definidas localmente: **a**, **b** y **ret**

Analizando lo que hemos aprendido vemos que la función main no puede acceder a num1, num2 ni ret. Y de igual manera la función max no puede acceder a a, b y ret. Para que main pueda decirle a max cuáles son los valres a comprar tiene que pasar sus variables locales como argumentos al llamarla max(a, b).

Esto lo que generará es que al iniciar el ámbito local de la función max, se copiará **los valores** a sus variables locales establecidas como parámetros num1 y num2. El orden es lo que definirá qué argumento cae en qué parámetro.

En la declaración ➔ max(int num1, int num2)

En la invocación ➔ max(a , b)

Es por esta razón que las variables al ser de ámbitos distintos, podrían llamarse igual, o diferente, sin importar al ejecutarse.

Veamos cómo sería la definición de una función para calcular el promedio de notas en nuestro algoritmo:

```
1   float calcularPromedioNotas(int nota1, int nota2) {
2           int promedio = 0;
3       promedio = (nota1 + nota2) / 2;
4           return promedio;
5   }
6
7   int main() {
8           int prom = 0;
9           int primerNota, segundaNota;
10
11          // Obtener Notas de Un alumno
12
13          prom = promedio(primerNota, segundaNota);
14
15          // Mostrar Promedio de Notas
16
17  }
```

Vemos que aún sin saber cómo se realizarán los módulos Obtener Notas de Un Alumno, ni Mostrar Promedio de Notas, ya podemos completar la parte que sí conocemos y definimos. Este también es un beneficio de la Modularización de los algoritmos, me permite poder avanzar por partes sin tener claras las implementaciones aún.

Procedimientos

Analicemos en el ejemplo anterior los módulos restantes:

Obtener Notas de Un alumno: Sabemos sólo por el nombre qué es lo que debería realizar, cuál es el objetivo de dicho módulo. Tomar las dos notas del alumno.

Pero si intentáramos realizarlo con una función, nos encontraremos con un inconveniente. Las funciones sólo devuelven un único resultado, de hecho, así las habíamos definido. Por lo cual no podemos hacer una función que devuelva dos notas, es decir esto es **completamente inválido:** *int, int obtenerNotas()*

Para estos casos existe el otro tipo de módulo, que se llama Procedimiento.

Definición	Un Procedimiento es un módulo que puede devolver **uno, muchos o incluso ningún resultado**

Los procedimientos, al igual que las funciones utilizan los parámetros para comunicarse con otros módulos, la diferencia es que la salida de un procedimiento no está limitada a un solo valor como en las funciones, sino que pueden ser muchos o incluso ningún valor.

Parámetros de Salida / Pasaje por Referencia

Para solucionar el problema que planteamos (tomar dos notas y devolverlas al programa principal) definiremos un procedimiento que tiene dos “parámetros de salida”.

Los parámetros de salida se declaran casi igual que los parámetros que hemos usado en las funciones (y que eran de entrada) solamente se debe agregar el símbolo & al parámetro. Veamos cómo será la definición:

void obtenerNotas (int **&**nota1, int **&**nota2);

El tipo de retorno es void dado que el procedimiento no retorna ningún valor a diferencia de la función, y los parámetros en lugar de ser contenedores de valores de entrada, ahora son **referencias** a variables y por lo tanto debemos especificar el símbolo &.

Si releemos la definición de los Ámbitos de variables recordaremos que las variables pertenecen a un ámbito específico y sólo las instrucciones de ese ámbito pueden acceder a sus valores. Entonces ahora lo que agregamos es que si deseamos que algún módulo modifique **una variable que no le pertenece** lo que deberá recibir como parámetro ya no es el valor de la variable sino la **referencia a la variable a modificar**. Esto en C/C++ es declarado con el símbolo &

Veamos cómo queda ahora el código de nuestro ejemplo:

```
1   float calcularPromedioNotas(int nota1, int nota2) {
2           int promedio = 0;
3       promedio = (nota1 + nota2) / 2;
4           return promedio;
5   }
6
7   void obtenerNotas(int &val1, int &val2) {
8           cout << “Ingrese primer nota:”;
9           // Por ser pasado por referencia el valor modificado es el de la variable
10          //primerNota del main
11          cin >> val1;
12
13          cout << “Ingese segunda nota:”;
14          cin >> val2;
15  }
16
17  int main() {
18          int prom = 0;
19          int primerNota, segundaNota;
20
21          obtenerNotas(primerNota, segundaNota);
22
23          prom = promedio(primerNota, segundaNota);
24
25          // Mostrar Promedio de Notas
26
27  }
```

Lo que recibe el procedimiento obtener notas ya no es una copia de los valores de las variables primerNota y segundaNota, sino que recibe sus **referencias** para que el módulo pueda modificarlas.

Se debe tener especial cuidado en el uso de este tipo de parámetros dado que las acciones ejecutadas en el módulo tendrán impacto en el estado de las variables del algoritmo que lo llamo, es decir se pierde un poco de la protección de datos que habíamos mencionado en las funciones.

Parámetros de Entrada / Pasaje por Valor

El único módulo que nos resta programar en nuestro ejemplo es el de *Motrar Promedio de Notas*, que deberá informar por consola el resultado obtenido por el módulo **promedio** por lo tanto sabemos que precisará un dato de entrada. Los parámetros de entrada son los que hemos usado en toda la explicación de funciones, por lo cual ya sabemos como funcionan, se copia el valor de la variable pasada como argumento a una variable local del módulo que la recibe.

La única diferencia que veremos entre un Procedimiento que sólo recibe parámetros de entrada y una función, es que una función retorna un dato, mientras que este procedimiento no debe retornar nada. Es decir el módulo Mostrar Promedio de Notas no necesita devolver ningún valor

al programa principal, solamente imprimir por consola, por esto diremos que es un Procedimiento y no una función:

void mostrarPromedio (float promedio);

El parámetro promedio será un parámetro de entrada por lo cual en él recibiremos el valor de la variable usada en la invocación. Cualquier acción que hagamos sobre su valor, no modificará el valor en el programa principal (igual que en las funciones).

```
1   float calcularPromedioNotas(int nota1, int nota2) {
2          int promedio = 0;
3       promedio = (nota1 + nota2) / 2;
4          return promedio;
5   }
6
7   void obtenerNotas(int &val1, int &val2) {
8          cout << “Ingrese primer nota:”;
9          // Por ser pasado por referencia el valor modificado es el de la variable
10         //primerNota del main
11         cin >> val1;
12
13         cout << “Ingese segunda nota:”;
14         cin >> val2;
15
16         return;
17  }
18
19  void mostrarPromedio(float promedio) {
20         cout << “El promedio del alumno es:” << promedio << endl;
21         return;
22  }
23
24
25  int main() {
26         int prom = 0;
27         int primerNota, segundaNota;
28
29         obtenerNotas(primerNota, segundaNota);
30
31         prom = promedio(primerNota, segundaNota);
32
33         mostrarPromedio(prom);
34  }
```

El procedimiento mostrarProcedimiento simplemente imprime por consola el valor recibido. En este caso hemos agregado la sentencia return; para marcar la finalización del procedimiento, como es void no se especifica ni valor ni variable, simplemente se fuerza la finalización. Esto puede ser utilizado en ciclos, condiciones, etc. Para forzar la finalización del procedimiento.

Combinando todo

Existe algunos casos donde podemos necesitar que un procedimiento reciba parámetros de entrada, pero también al tener más de una salida necesitará tener parámetros de salida. Esto lo podemos hacer simplemente combinando ambos tipos de parámetros en la declaración.

Veamos un ejemplo, supongamos que deseamos un procedimiento que realice una división entre dos números y calcule la parte entera y el resto.

```
1   void division (int dividendo, int divisor, int &cociente, int &resto) {
2           cociente = dividendo / divisor;
3           resto = dividendo % divisor;
4   }
5
6   int main() {
7           int divisor, dividend, c, r;
8           cout << “Ingrese el dividendo” << endl;
9           cin >> dividendo;
10          cout << “Ingrese el divisor” << endl;
11          cin >> divisor;
12
13          division(dividendo, divisor, c, r);
14
15          cout << “La división entera es:” << c << endl;
16          cout << “El resto de la división es:” << r << endl;
17  }
```

Ahora supongamos que queremos agregar la validación de que no se puede dividir por 0, entonces tenemos dos alternativas: o ponemos cociente y resto en 0 o lo informamos a través de algún retorno que indique si se pudo realizar la operación. Si así es, aunque sea un procedimiento también puede tener un valor de retorno, entonces si lo hacemos de esa forma la función quedaría

```
1   bool division (int dividendo, int divisor, int &cociente, int &resto) {
2           if (divisor == 0) {
3           return false;
4           }
5           cociente = dividendo / divisor;
6           resto = dividendo % divisor;
7           return true;
8   }
9
10  int main() {
11          int divisor, dividend, c, r;
12          cout << “Ingrese el dividendo” << endl;
13          cin >> dividendo;
14          cout << “Ingrese el divisor” << endl;
15          cin >> divisor;
16
17          if (division(dividendo, divisor, c, r)) {
18              cout << “La división entera es:” << c << endl;
19              cout << “El resto de la división es:” << r << endl;
20          } else {
21              cout << “No se pudo realizar la división” << endl;
22          }
23  }
```

Consejos para Modularizar

La técnica de modularizar para poder resolver un problema siempre es de gran utilidad y la debemos utilizar aunque luego no codifiquemos módulos independientes. Es decir, al plantearnos cómo resolver un problema siempre debemos “pensar en módulos”, luego dependiendo de si es de utilidad llevaremos el código a un módulo independiente o al algoritmo principal.

Por ejemplo, revisemos nuestro algoritmo para el cálculo de los promedios, desde un inicio identificamos y pensamos en 3 módulos. Obtener las notas, Calcular Promedio, Imprimir el Promedio. En consecuencia realizamos 2 procedimientos y 1 función para tener esos módulos codificados de forma independiente al programa principal. Para fines didácticos esto estuvo muy bien, pero si analizamos los dos procedimientos que generemos (Obtener notas e Imprimir Promedio) no nos significaron mucha ganancia a nivel de abstracción y reusabilidad.

El módulo Obtener notas, para que imprima mensajes que tengan sentido para el usuario quedo menos abstracto de lo que necesitaríamos para poder reutilizarlo, es decir no me va a servir para tomar cualquier dato, solo para tomar notas. De igual forma el módulo Imprimir promedio no me aporta mucho más que sacar las sentencias de impresión por consola, de igual forma el mensaje es muy específico como para que ese módulo sea reutilizado.

En cambio, la función promedio, si podremos reutilizarla siempre que deseemos calcular el promedio entre dos números. Para esto es muy importante que **tanto el nombre de la función como los nombres de parámetros no sean específicos del problema que estamos resolviendo sino lo más generales posibles**, si por ejemplo hubiéramos hecho la función

float promedioNotas(int nota1, int nota2)

operativamente es equivalente pero es menos abstracta dado que si deseamos calcular el promedio entre dos valores que representan distancias quedaría por lo menos raro.

Otro punto a considerar es que los módulos deben hacer una sola cosa para potenciar la abstracción y la reusabilidad. Por ejemplo, a alguien podría habérsele ocurrido hacer un procedimiento que además de obtener las notas ya hiciera la acumulación y la retornara, pero de esa forma nos hubiéramos perdido la posibilidad de usar un módulo genérico y reutilizable como es la función **promedio**.

Recursividad

¿Qué es la recursividad? La respuesta simpe es, cuándo una función se llama a sí misma. ¿Pero cómo pasa esto? Por qué pasaría esto, ¿y qué usos tiene?

Cuando hablamos de recursividad, en realidad estamos hablando de crear un bucle de repetición. Empecemos viendo un bucle básico

```
for(int i=0;  i<10; i++) {
    cout << "El número es: " << i <<
    endl;
}
```

Ya deberíamos saberlo, pero por las dudas aclaremos, esto debería generar la siguiente salida:

```
El número es: 0
El número es: 1
El número es: 2
El número es: 3
El número es: 4
El número es: 5
El número es: 6
El número es: 7
El número es: 8
El número es: 9
```

Ahora modifiquémoslo para que, en lugar de imprimir directamente, llame a una función que se encargue de eso

```
1  #include <iostream>
2  using namespace std;
3
4  void numberFunction(int i) {
5    cout << "El número es: " << i << endl;
6  }
7
8  int main() {
9
10     for(int i=0; i<10; i++) {
11       numberFunction(i);
12     }
13     return 0;
14 }
15
```

Declaramos un procedimiento (porque no tiene valor de retorno), que recibe un parámetro de entrada entero “int i” y lo único que hace es imprimir por consola el número recibido con la misma sentencia que antes estaba dentro del For. La función es llamada mientras el valor de i sea menor que 10.

Usando recursividad ya no necesitamos el ciclo For en el programa principal, porque haremos que nuestra función se llame a sí misma siempre que el número sea menor que 10 (misma condición que antes).

```
1  #include <iostream>
2  using namespace std;
3
4  void numberFunction(int i) {
5    cout << "El número es: " << i << endl;
6    i++;
7    if(i<10) {
8         numberFunction(i);
9    }
10 }
11 int main() {
12        int i = 0;
13        numberFunction(i);
14        return 0;
15 }
```

Vemos que, aunque hay una sola llamada a la función 'numberFunction' en el main, la función se invocará siempre que el valor de i recibido sea menor que 10.

Esto se produce porque cada llamada que se realiza desde la misma función se irá apilando, y al llegar a la llamada en la cual el valor de i es 10, no seguirá apilando llamadas y comenzará a volver el control de programa a la función que lo llamo, hasta llegar nuevamente al main.

Los algoritmos recursivos siempre tienen 2 partes, una es la regla trivial o caso base, que es lo que permite realizar el corte de la recursiviad, y luego la llamada a la misma función, pero con alguna modificación en sus parámetros.

Concepto de recursividad:

Es un proceso que se basa en su propia definición. Una función puede invocarse a sí misma como parte de los tratamientos de cálculo que necesita para hacer su tarea

Parte de instancias complejas y las define en términos de instancias más simples del mismo problema, llegando a un punto donde las instancias más simples son definidas explícitamente.

Define el problema en términos de un problema más simple de la misma naturaleza. Debe disminuir el espacio del problema en cada llamada recursiva

Hay una instancia particular que se conoce como caso base o caso degenerado

Divide el problema original en subproblemas más pequeños. Cuando es lo suficientemente chico se resuelve directamente y se combinan soluciones del subproblema hasta que queda resuelto el problema

Tiene:

- Una ecuación de recurrencia, en función de términos anteriores Tn = F(Tn-1, Tn-2, T0).
- Uno o varios términos particulares que no dependen de los anteriores. Ti = G(i) (base)

Veamos algunos ejemplos de problemas que pueden resolverse recursivamente

Función Factorial

- Ecuación de recurrencia : n! = n * (n-1)!
- Condiciones particulares: 0! = 1

```
1 double factorial(int n)
2 {
3        if (n==0)                    // Condiciones particulares: 0! = 1
4         return 1;
5        else
6          return n * factorial(n-1); // Ecuación de recurrencia :  n! = n * (n-1)!
7 }
```

Función PotenciaNatural

- Ecuación de recurrencia : an = a (n-1) * a si n > 1
- Condiciones particulares: a0 = 1

```
1 double potencia(int b, int p)
2 {
3         if(p == 0) {
4         return 1       // Condiciones particulares a0 = 1
5         } else {
6     return base * potencia(base, exponente -1)  // Ecuación de recurrencia
7 }
```

Sucesión de Fibonacci

Una pareja de conejos tarda un mes en alcanzar la edad fértil y a partir de aquí un mes en engendrar otra pareja que al alcanzar la fertilidad engendraran otra pareja, entonces ¿Cuántos conejos habrá al término de N meses? Si los animales no mueren cada mes hay tantas parejas como la suma de los dos meses anteriores

- Ecuación de recurrencia : Fib(n) = Fib(n-1) + (n-1)!
- Condiciones particulares: Fib(0) = 1 y Fib (1) = 1

Meses	Padres	Hijos	Nietos	Cant Parejas
0	I			1
1	M			1
2	M	I		2
3	M	M I		3
4	M	M M I	I	5
5	M	M M M I	M I I	8

¿Cuál sería su algoritmo?

Unidad Temática N°7

~ Vectores y Estructuras ~

"Los malos programadores se preocupan por el código. Los buenos programadores se preocupan por las estructuras de datos y sus relaciones."

~ Linus Torvalds

Vectores y Estructuras

En esta unidad nos proponemos presentarte nuevas estructuras de datos para almacenar varios elementos. Te explicaremos qué son los vectores y las matrices y cómo utilizarlos. Además, te mostraremos qué son las Estructuras de Datos.

En las unidades anteriores trabajamos almacenando los datos de los programas en variables y constantes. Estos identificadores tienen la particularidad de que almacenan un único valor. Las variables y constantes de tipos de datos simples son muy útiles, pero hay algoritmos más complejos en los que necesitarás guardar múltiples datos. Es por eso que serán necesarias otras estructuras de datos para poder trabajar con muchos valores y de distintos tipos de datos.

En esta unidad, como anticipamos, veremos cómo trabajar con vectores, matrices y estructuras de datos

Vectores

Supongamos que queremos calcular el promedio de edades de los alumnos de un curso. Seguramente no tendrás ningún problema en resolverlo, ya que deberás realizar un programa en el que el usuario vaya ingresando las edades, y por cada edad ingresada se deberá acumular y contar. De esta manera tendrás la suma total de las edades y la cantidad de alumnos, datos necesarios para obtener el promedio.

Ahora bien, le agregaremos un resultado más: queremos saber cuántos alumnos superan la edad promedio.

¿Cómo realizarlo?

Una vez que obtuviste el promedio de edades, deberás comparar cada edad para determinar si supera la edad promedio, y en caso afirmativo, contarla como una edad que supera la edad promedio.

Pero, ¿Cómo hacerlo si los datos que ingresó el usuario ya no los tenés? ... no los pudiste guardar porque no sabías la cantidad de alumnos, puede ser 10, 50 o un curso de 500.

Si la cantidad de alumnos es, por ejemplo, 500:

¿Utilizarás 500 variables para almacenar los datos de 500 alumnos?

Te estarás imaginando que esta no es una solución práctica. Por lo tanto necesitarás alguna otra estructura de datos para poder almacenar todas las edades de los alumnos y poder consultarlas nuevamente. Es en este momento donde te presentamos a los arreglos.

¿Qué es un vector?

Un vector puede guardar varios datos asociados a un mismo nombre. Es una variable que, en lugar de tener un único valor, tiene asociado una cantidad finita de valores. ¿Y por qué finita? Porque al crear un arreglo debés indicar cuál es la cantidad de elementos que vas a guardar en él.

Los vectores también se denominan arreglos unidimensionales o arrays.

Veamos un ejemplo....

Para definir un arreglo, como te contamos antes, necesitás conocer la cantidad de elementos que tendrá. De ahí surge que un arreglo es una estructura de datos estática, ya que siempre tiene la misma cantidad de elementos.

Declaración de un vector

```
int vec[4];
```

Como dijimos antes un vector es una estructura que nos permite con el nombre de una variable acceder a múltiples valores almacenados, en este ejemplo sencillo declaramos un vector llamado ‘vec’ que podrá guardar hasta cuatro valores de tipo entero.

Podemos definir un vector de la siguiente manera:

Es un conjunto finito de elementos del mismo tipo de dato y naturaleza que ocupan posiciones contiguas de memoria y se encuentran asociados a un mismo nombre en común.

Analicemos la definición de arreglo...

• Conjunto finito: La cantidad de elementos de un arreglo es finita y se conoce en el momento de definir el arreglo. No se pueden agregar o eliminar elementos, ya que un arreglo tiene siempre la misma cantidad de componentes.

• Elementos del mismo tipo de dato y naturaleza: Todos los elementos son del mismo tipo de dato. Por ejemplo, si vamos a utilizar un arreglo para guardar edades de personas, todos los elementos serán del tipo de dato ENTERO

• Ocupan posiciones contiguas de memoria: Los elementos se almacenan en la memoria uno a continuación de otro. Como cada uno de los elementos ocupa el mismo espacio, se puede acceder a cualquier elemento conociendo la posición de memoria del primer elemento.

• Asociados a un mismo nombre: Un arreglo es una VARIABLE, por lo tanto, tendrá un solo nombre que estará asociado a un conjunto de valores. A cada uno de los elementos se accede directamente a través de su posición, utilizando lo que llamaremos el Índice.

Declaración de vectores

Para poder declarar un vector es necesario conocer la cantidad de elementos máxima que tiene que almacenar, esto también lo llamamos dimensión y es el valor que se pasa entre corchetes.

int vec[**4**]

Pero podría pasar que esa cantidad sea variable según algún dato ingresado por el usuario o calculado por el algoritmo, con lo cual tendremos que usar una variable en la declaración que nos dirá cuál es la dimensión:

int vec[**N**]

Lo importante y que no se debe olvidar nunca es que **la variable N debe tener el valor** al momento de usarla en la declaración del vector, es decir lo este código es **incorrecto**:

```
1  int main() {
2      int N;
3      int edades[N];  //Define el vector de N elementos pero N no tiene valor!!!
4      cout << “Ingrese la cantidad de edades”;
5      cin >> N;
6  }
7
```

Ya que N no tiene el valor ingresado en la línea 3 al momento de la declaración del vector. La forma correcta sería declarar el vector luego de que se lee el valor en N:

```
1  int main() {
2      int N;
3      cout << “Ingrese la cantidad de edades”;
4      cin >> N;
5
6      int edades[N];
7  }
```

También es posible declarar e inicializar el vector utilizando cualquiera de las siguientes dos formas:

```
1 int edades[] = {1, 2 ,3 ,4};
```

```
int edades[4] = {1, 2, 3, 4}:
```

En la primera no es necesario especificar la dimensión dado que se infiere de la cantidad de elementos usados en la inicialización.

Acceso a cada elemento

Índice

Como dijimos previamente, todos los elementos del vector están asociados a un mismo nombre, pero tenemos una manera de acceder a cada uno de los elementos de forma individual, y eso se hace a través del índice.

Definición	El índice es la **posición relativa** que tiene cada elemento **en el arreglo**.

Posición inicial

En C/C++ y en todos los lenguajes de programación el primer elemento se encuentra en la posición 0, es decir que para acceder al primer elemento debemos acceder a la posición 0; para acceder al segundo elemento accederemos a la posición 1 y así sucesivamente.

Acceso directo

Se puede acceder directamente a cada elemento particular por medio del índice sin necesidad de tener que pasar por los elementos anteriores. Esto es lo que llamamos acceso directo a cada elemento.

Para especificar la posición del vector a la que queremos acceder, tanto para leer como para guardar un valor utilizaremos su posición encerrada entre corchetes [ ]:

```
edades[POS]
```

Donde POS es la posición del elemento al que queremos acceder. Recordemos que se tienen que cumplir las siguientes condiciones:

POS >= 0 Y POS < N

Veamos un ejemplo para entender el acceso a cada elemento

```
1 edades[0] = 20;
2 edades[1] = 21;
3 cout << edades[1];
```

1. En la primera acción asignamos el valor 20 al elemento de la posición 0.
2. En la segunda acción asignamos el valor 21 al elemento de la posición 1
3. En la tercera acción mostramos el valor de la posición 1.

¿Qué valor se mostrará por pantalla?

El número **21**.

Cuando trabajamos con vectores es fundamental poder identificar la posición y el valor del vector en la posición. El elemento está relacionado por la posición que ocupa (el índice), mientras que el valor es el contenido (el dato) que se encuentra almacenado en esa posición.

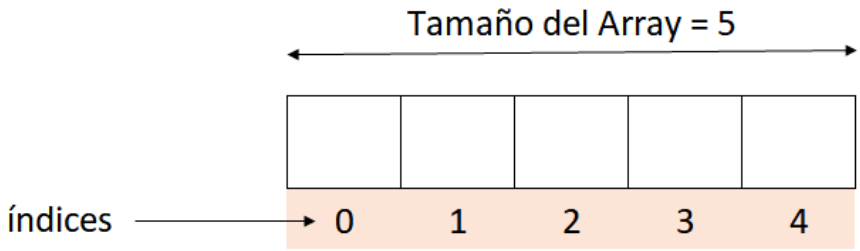
En el ejemplo tenemos que el elemento de la posición 0 tiene el valor 20, y el elemento de la posición 1 tiene el valor 21.

Representación gráfica

Siguiendo con el ejemplo del vector de edades, si tuviéramos la siguiente declaración:

```
1 int edades[5];
```

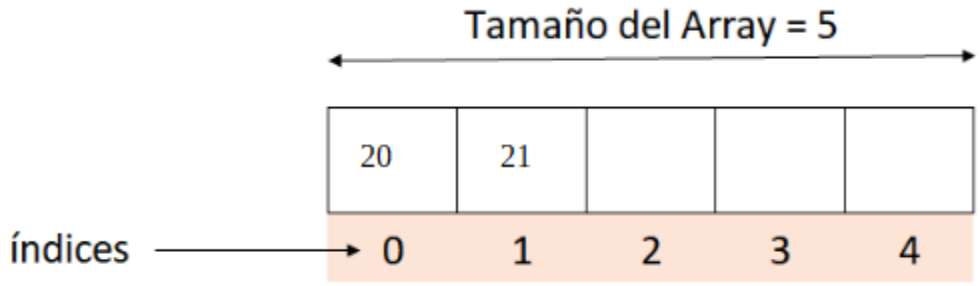
Gráficamente el arreglo definido lo podemos ver de la siguiente manera:



Luego de las sentencias:

```
1 edades[0] = 20;
2 edades[1] = 21;
```

Resulta



En este ejemplo se puede decir que:

• Edades sub 1 es igual a 21.

• El valor del elemento de la posición 1 es 21.

• Si bien tenemos 5 elementos en EDADES, al haber cargado valores para los dos primeros, diremos que el arreglo tiene 2 elementos por lo cual hay 3 posiciones “libres”. Se debe tener especial cuidado con las posiciones libres ya que si no han sido inicializadas esas posiciones podrían tener basura.

• Las posiciones con datos válidos son la 0 y la 1, por lo cual esas son las posiciones que accederemos en caso que queramos leer valores.

Algoritmos básicos: Cargar Vector

Veamos un algoritmo que utilizaremos para cargar un arreglo con los datos que ingresa un usuario por teclado

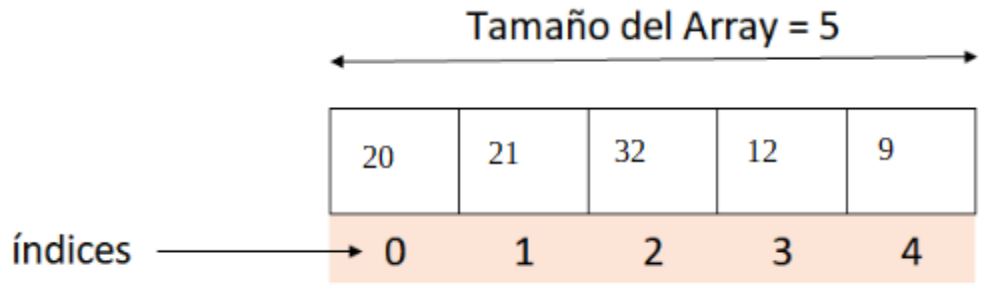
```
1  int main() {
2      int N = 0;
3      cout << “Ingrese cantidad de edades”;
4      cin >> N;
5      int edades[N];
6      for(int i = 0; i < N; i++) {
7           cout << “Ingrese una edad: “;
8           cin >> edades[i];
9       }
   }
```

Analicemos el algoritmo:

* Lo primero que hacemos es averiguar la cantidad de elementos que deberemos almacenar. En este caso esa cantidad es provista por el usuario.

* Luego declaramos el vector de **N elementos**, tener en cuenta que **N ya debe tener un valor al realizar la declaración del vector**.

* Por último generamos un ciclo que repetirá las sentencias para que el usuario ingrese una edad. En este caso queremos llenar el vector completo por lo cual el contador del for irá de 0 a N – 1. De esta forma en la primer iteración el valor ingresado será almacenado en edades[0] y el último en edades[N – 1]. Por ejemplo, si el usuario ingresara un N = 5 y luego las edades 20, 21, 32, 12 y 9 el vector al finalizar el algoritmo quedaría:



Algoritmos básicos: Mostrar Vector

Ahora veremos una forma de mostrar todos los elementos junto con la posición en que se encuentran dentro del vector. Este algoritmo lo podremos utilizar cada vez que necesitemos mostrar todo un vector.

```
1   int main() {
2       int N = 0;
3       cout << “Ingrese cantidad de edades”;
4       cin >> N;
5       int edades[N];
6        for(int i = 0; i < N; i++) {
7            cout << “Ingrese una edad: “;
8            cin >> edades[i];
9        }
10       for(int i = 0; i < N; i++) {
11           cout << “Posicion: “ << i << “ Valor: ” << edades[i] ;
12       }
    }
```

En este algoritmo lo que vemos que se ha agregado es otro ciclo que vuelve a generar las posiciones del vector de 0 a N – 1 y por cada iteración imprimirá la posición y el valor del vector en la posición. La salida de este algoritmo si cargara los mismos datos antes enumerados sería:

>Posicion: 0 Valor: 20
>Posicion: 1 Valor: 21
>Posicion: 2 Valor: 32
>Posicion: 3 Valor: 12
>Posicion: 4 Valor: 9

Arreglos como parámetros

Como ya sabemos, los parámetros pueden ser pasados por Valor o por Referencia. Pero si desearamos pasar un vector por valor esto requeriría que se nos asigne nuevamente la cantidad de memoria necesaria para el vector que ya pedimos en el algoritmo principal, y además debería hacer el copiado de todos los bytes de un espacio al otro. Para evitar todo esto, lo que haremos es pasar los vectores por referencia SIEMPRE. De hecho C/C++ no permite otra forma de pasaje de vectores como parámetro.

Para especificar que un módulo recibe un vector, y nuevamente remarcamos lo que recibirá es la dirección de memoria del vector ya que es pasado por referencia, lo que haremos es especificar el tipo del vector, el nombre con el que lo referenciamos en el módulo y los corchetes vacíos [] para indicar que es un vector.

```
1 void funcion (int vector[]) {
2 }
```

Ahora pensemos, si quisiera en este procedimiento imprimir todos los elementos del vector, me encontraría con el problema de saber cuántos elementos hay. Por lo tanto además del vector siempre deberemos recibir la cantidad de elementos presentes para poder realizar el ciclo correctamente.

Veamos por ejemplo el procedimiento imprimir que haga lo mismo que teníamos antes en el algoritmo principal:

```
1 void imprimir(int vec[], int cant) {
2     for(int i = 0; i < cant; i++) {
3         cout << “Posicion: “ << i << “ Valor: “ << vec[i];
4     }
5 }
```

De esta forma el procedimiento recibe la referencia al vector por lo cual ya tiene los valores cargados y podrá recorrerlo e imprimirlo.

También podríamos hacer otro procedimiento que realice la carga:

```
1 void cargar(int vec[], int cant) {
2     for(int i = 0; i < cant; i++) {
3         cout << “Ingrese una edad”;
4         cin >> vec[i];
5     }
6 }
```

Ahora nuestro programa principal podría ser:

```
1 int main() {
2     int N;
3     cout << “Ingrese la cantidad de edades”;
4     cin >> N;
5     int edades[N];
6     cargar(edades, N);
7     imprimir(edades, N);
8 }
```

¿Cómo usamos los vectores?

El objetivo de utilizar los vectores o las distintas estructuras es permitirnos almacenar muchos datos de forma simple para su acceso. Con esto además el conjunto de datos puede separarse del algoritmo de su obtención y tratamiento. Es decir antes al solicitar las edades si queríamos saber cuál era la mayor debíamos averiguarlo en la misma carga de datos, ya que los datos luego no los tendríamos más. Esto generaba una mezcla de etapas como hemos dicho al inicio del apunte, todo algoritmo tiene entradas, procesos y salidas, pero en este caso debíamos hacer la entrada junto con algo de proceso.

Ahora podemos separar de forma más clara los procesos de las entradas, y esto redundará en una mejor modularización de nuestros algoritmos.

Por ejemplo el programa que pide N edades y averigua la mayor ahora lo podremos escribir usando una función buscaMax:

```
1 int buscaMax( int vec[], int cant) {
2     int max = vec[0];
3     for(int i = 1; i < cant; i++) {
4         if (vec[i] > max) {
```

```
5               max = vec[i];
6           }
7       }
8       return max;
9   }
```

Si analizamos el algoritmo es lo mismo que hubiéramos hecho antes de conocer los vectores, tomar el primer valor ingresado como máximo y luego por cada valor evaluar si es mayor que el máximo. La diferencia ahora es que para tomar el primer valor, usaremos el índice 0 y luego recorreremos el vector desde 1 (ya que el 0 lo tenemos como máximo) hasta la última posición del vector.

Otra alternativa mejor sería devolver no el valor máximo sino la posición del máximo, ya que con la posición se podrá acceder al valor:

```
1   int buscaPosMax( int vec[], int cant) {
2       int max = vec[0];
3       Int pos = 0;
4       for(int i = 1; i < cant; i++) {
5           if (vec[i] > max) {
6               max = vec[i];
7               pos = i;
8           }
9       }
10      return pos;
11  }
```

Otra función que podremos realizar ahora es promedio que reciba un vector de enteros y devuelva el promedio de sus elementos:

```
1   float promedio(int vec[], int n) {
2       int sum = 0;
3       for(int i = 0; i < n; i++) {
4           sum += vec[i];
5       }
6       return (float) sum / n ;
7   }
```

Ahora podremos hacer nuestro problema original, contar cuántos alumnos tienen una edad mayor a la edad promedio:

```
1   int main() {
2       int n = 0;
3       cout << “Ingrese cantidad de alumnos”;
4       cin >> n;
5       int edades[n];
6       cargar(edades, n);
7       float prom = promedio(edades, n);
```

```
8       int cant = 0;
9       for(int i = 0; i < n; i++) {
10          if (edades[i] > prom) {
11              cant++;
12          }
13      }
14      cout << “Hay ” << cant << “ alumnos mayores que el promedio”;
15  }
```

Matriz

En la sección anterior dijimos que los vectores también son llamados arreglos unidimensionales. Tal como vimos en la declaración y acceso a los valores del vector, el vector tiene una única dimensión. Pero existen arreglos de más dimensiones por lo cual los arreglos son clasificados en:

- Unidimensionales (vectores)
- Bidimensionales (tablas o matrices)
- Multidimensionales (tres o más dimensiones)

Una matriz es un arreglo en donde a cada elemento se accede por medio de dos índices.

Declaración

Al igual que con los arreglos unidimensionales, en este tipo de estructura de datos también debemos definir su tamaño antes de utilizarla, con el agregado de la segunda dimensión. De igual forma que antes con los vectores, al declarar la matriz colocaremos entre corchetes las dimensiones, siendo la primera lo que denominaremos como FILAS y la segunda como COLUMNAS.

```
1  int mat[3][4]
```

Esta matriz declarada tendrá 3 filas y cuatro columnas. Pero también es posible que necesitemos que alguna de las dimensiones sea ingresada por el usuario o calculada, para esto podremos utilizar variables al momento de la declaración. Pero al igual que con los vectores estas variables deben tener valor al momento de usarlas para la declaración:

`N filas por 4 columnas`	`3 filas por N columnas`	`N filas por M columnas`

`int mat[N][4]`	`int mat[3][N]`	`int mat[N][M]`

Cabe volver a remarcar que tanto en los vectores como en las matrices los elementos deben ser siempre del mismo tipo. En los ejemplos anteriores, todos los elementos guardados en la matriz serán enteros.

También es posible realizar la inicialización con valores de la matriz de manera similar a los vectores:

```
int mat[2][3] = {{2, 4, 3}, { 4, 5, 6}};
```

Acceso a cada elemento

Índice

En el caso de las matrices, como es un arreglo con dos dimensiones, a cada elemento se accede por medio de dos índices, el primero corresponde a la fila y, el segundo, corresponde a la columna.

Veamos un ejemplo para entender el acceso a cada elemento:

```
1 MAT[0][2] = 20
2 MAT[1][4] = 21
3 cout << MAT[1][4]
```

1. En la primera acción, asignamos el valor 20 al elemento de la fila 0, columna 2.
2. En la segunda acción asignamos el valor 21 al elemento de la fila 1, columna 4.
3. En la tercera acción mostramos el valor del elemento de la fila 0, columna 2.

¿Qué valor se mostrará por pantalla?

El número 21.

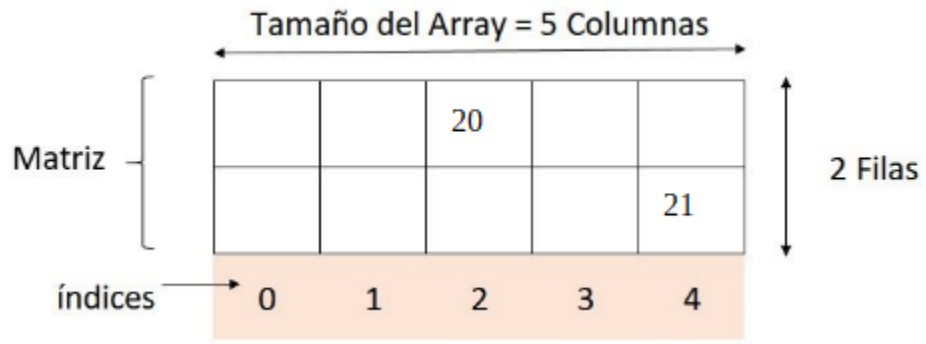
La posición de la fila y la columna se indica por medio de los dos números, teniendo en cuenta que el primero corresponde a la fila y, el segundo, a la columna.

Representación gráfica

Si tuviésemos la siguiente matriz y asignaciones:

```
1 int mat[2][5];
2 mat[0][2] = 20;
3 mat[1][4] = 21
```

Gráficamente la matriz definida la podemos ver de la siguiente manera:



• MAT sub 0, 2 es igual a 20.

• El valor del elemento de la posición de la fila 0 y columna 2 es 20.

Algoritmos básicos: Cargar Matriz

Veamos el algoritmo que utilizaremos para cargar una matriz con los datos que ingresa un usuario por teclado.

```
1  int main() {
2      int N = 3;
3      int M = 4;
4      int mat[N][M];
5      for(int i = 0; i < N; i++) {
6          for(int j = 0; j < M; j++) {
7              cout << “Ingrese un valor”;
8              cin >> mat[i][j];
9          }
10     }
11 }
```

En este ejemplo se cargan todas las posiciones de matriz y el orden de carga es por filas. Es decir que el primer elemento que ingrese el usuario se cargará en la posición fila 0, columna 0. El segundo elemento que ingrese el usuario se cargará en la posición de la fila 0, columna 1. Y así sucesivamente, recorriendo la matriz primero por la fila 0, cuando llegue al último elemento de la fila 0, se pasará a la fila 1 .... Y así hasta recorrer y cargar toda la matriz.

Si en lugar de cargar por filas desearamos cargarla por columnas lo único que debemos hacer es generar en el for externo las columnas en lugar de las filas. La forma más clara y sencilla de hacerlo es simplemente intercambiar los for:

```
1   int main() {
2       int N = 3;
3       int M = 4;
4       int mat[N][M];
5       for(int j = 0; j < M; j++) {
6           for(int i = 0; i < N; i++) {
7               cout << “Ingrese un valor”;
8               cin >> mat[i][j];
9           }
10      }
11  }
```

De esta forma por cada valor de j que representa una columna se generan todos los valores de i que representan las filas.

Lo más importante a tener en cuenta es siempre que los **índices no excedan el valor de la dimensión** que tienen que iterar, es decir si decimos que i iterara las filas, entonces no puede superar el valor usado al definir la matriz como filas, en este caso N; y lo mismo con la j si la usaremos para las columnas.

Algoritmos básicos: Mostrar Matriz

Del mismo modo que al cargar una matriz se puede recorrer por filas o por columnas, el mostrar los elementos de una matriz también podemos hacer ambos recorridos.

Veamos ahora una rutina que muestra los elementos de una matriz, recorriéndola por columnas. Es decir que el primer elemento que se muestra será el de la fila 0 columna 0, luego el de la fila 1, columna 0, y cuando finalice la columna 0 se pasará a la columna 1. Y así hasta llegar hasta la última columna.

```
1   int main() {
2       int N = 3;
3       int M = 4;
4       int mat[N][M];
5       for(int j = 0; j < M; j++) {
6           for(int i = 0; i < N; i++) {
7               cout << mat[i][j];
8           }
9       }
10  }
```

Matrices como parámetros

Al igual que sucede con los vectores, las matrices sólo pueden pasarse como parámetros por Referencia. Pero al utilizarlas en C/C++ tendremos una limitante más.

Intuitivamente creeríamos que para declarar una matriz como parámetros haríamos algo como:

```
1  void cargarMatriz(int mat[][], int filas, int cols) {
2      for(int i = 0; i < filas; i++) {
3              for(int j = 0; j < cols; j++) {
4                  cin >> mat[i][j];
5              }
6      }
7  }
```

Pero el compilador admite que sólo haya una dimensión variable, como cuando lo usamos para vectores, y en este caso tenemos dos dimensiones. Por lo cual deberemos especificar al menos una de las dimensiones en la declaración del procedimiento:

```
1  void cargarMatriz(int mat[][3], int filas) {
2      for(int i = 0; i < filas; i++) {
3              for(int j = 0; j < 3; j++) {
4                  cin >> mat[i][j];
5              }
6      }
7  }
```

De esta forma la dimensión variable son las filas y podremos tener N filas, pero siempre serán solamente 3 columnas.

Una forma de paliar esta limitación es usar valores máximos de filas y columnas, de esta forma podremos tener la matriz de N x M siendo N < **MAX_FILAS** y M < **MAX_COLS** estos valores máximos no pueden ser ni variables ni constantes de las que usamos comunmente sino que deben ser valores definidos para que estén disponibles en tiempo de compilación, esto lo logramos usando la clausula #define:

```
1  #define MAX_FILAS 100
2  #define MAX_COLS 100
3
4  void cargarMatriz(int mat[MAX_FILAS][MAX_COLS], int filas, int cols) {
5      for(int i = 0; i < filas; i++) {
6              for(int j = 0; j < cols; j++) {
7                  cin >> mat[i][j];
8              }
9      }
10 }
11
```

```
12  int main() {
13      int mat[MAX_FILAS][MAX_COLS];  //Ya defino la matriz con los valores maximos
14      int N, M;
15      cout << “Ingrese cant de filas”;
16      cin >> N; //Tomo la cantidad de filas a trabajar, debe ser < MAX_FILAS
17      cout << “Ingrese cant de columnas”;
18      cin >> M; // Tomo la cantidad de columnas a trabajar, debe ser < MAX_COLS
19      cargarMatriz(mat, N, M);
20      imprimirMatriz(mat, N, M);
21  }
```

Este algoritmo es más parecido al que habíamos intentado al principio, el único inconveniente será el desperdicio de memoria en los casos en que **N** es mucho menor a **MAX_FILAS** y **M** es mucho menor que **MAX_COLS**.

Veamos un ejemplo completo. Realicemos un programa que permita al usuario cargar N alumnos y por cada alumno debe poder ingresar el legajo, y las tres notas de la materia. Vemos que con un vector no nos alcanzaría para almacenar toda esta información, necesitaríamos al menos 4 vectores: uno para los legajos, y uno más por cada nota. En cambio utilizando matrices podremos definir una matriz de N x 4.

Como sabemos que la cantidad de columnas es fija, podremos usar la forma mat[N][4] tanto en la declaración como en las funciones si hicieran falta.

	0 (Legajo)	1 (Nota 1)	2 (Nota 2)	3 (Nota 3)
0	12323	6	3	8
1	22112	4	8	7
2	33223	6	2	9
N	42322	7	2	10

La función cargarAlumnos permitirá cargar todos los datos de cada alumno, primero el legajo y luego cada nota. La función podría ser de la siguiente forma:

```
1   void cargarAlumnos(int mat[][4], int cant) {
2       for(int i = 0; i < cant; i++) {
3           cout << “Ingrese el legajo del alumno”;
4           cin >> mat[i][0];   //En la primer columna de cada fila estará el legajo
5
6           for(int j = 1; j < 4; j++) {
7               cout << “Ingrese la nota “ << j << “: ”;
8               cin >> mat[i][j];
9           }
10      }
11  }
12
```

Si quisiéramos imprimir un listado de la siguiente forma:

Alumno:
- Nota 1:
- Nota 2:
- Nota 3:
- Promedio:

El algoritmo podría ser:

```
1   void imprimirListado(int mat[][4], int cant) {
2       int sum = 0;
3       for(int i = 0; i < cant; i++) {
4           cout << “Alumno: ” << mat[i][0];
5           sum = 0;
6           for(int j = 1; j < 4; j++) {
7               cout << “    - Nota “ << i << “: “ << mat[i][j];
8               sum += mat[i][j];
9           }
10          cout << “    - Promedio: ” << (float) sum / 3;
11      }
12  }
```

Y el algoritmo principal que llama a nuestros procedimiento:

```
1   int main() {
2       int N;
3       cout << “Ingrese cantidad de alumnos”;
4       cin >> N;
5       mat[N][4];  //Solo cuando ya tengo la cantidad de alumnos puedo declarar mat
6
7       cargarAlumnos(mat, N);
8       imprimirListado(mat, N);
9
10  }
```

Estructura de Datos

Volviendo a nuestro ejemplo de edades de alumnos, supongamos que ahora además de pedir la edad de un alumno, también queremos saber el nombre y la cantidad de materias aprobadas. Es decir que **por cada alumno tendremos tres valores** que guardaremos en tres variables distintas. En el caso en que se necesite hacer un procedimiento que imprima los datos de cada alumno, tendrá que tener esos **tres datos como parámetros**.

Hasta ahora no habrás encontrado ninguna dificultad en este problema, pero qué sucede si te decimos que, además de los datos mencionados, también queremos guardar otros valores, como, por ejemplo, el número de documento, el sexo, la cantidad de materias que está cursando.... Tendrás que agregar todos esos datos en la lista de parámetros de cada módulo que

quiera trabajar con los datos de un alumno. Como habrás notado, se estará haciendo más complejo el algoritmo, y hasta cierto punto, difícil para entenderlo y hasta para probarlo.

Evidentemente necesitamos algo para poder guardar de forma más apropiada estos valores. Es en este momento donde necesitamos utilizar las **Estructuras** o Registros de datos.

Veamos cómo definimos un registro para guardar los datos de un alumno en C++:

```
1  struct Alumno {
2      string Nombre;
3      int Edad;
4      int Aprobadas;
5  };
6
7  int main() {
8      Alumno alum;
9  }
```

Para definir un Registro debemos definir un nuevo tipo de dato que indique los elementos que contiene y cuáles serán sus nombres.

En este ejemplo estamos definiendo un nuevo tipo de datos que se llama **Alumno** como un Registro. Este Registro tiene tres campos que son: **Nombre**, de tipo de dato String; **Edad**, de tipo de dato Entero y **Aprobadas** que también es de tipo de dato Entero.

Luego definimos una variable que se llama **alum** que es un **Registro del tipo de dato Alumno**. Es decir, que la variable alum guardará tres valores bajo su mismo nombre lógico.

Uno de los principales motivos por el que definimos Registros es para asociar bajo una misma variable a un conjunto de elementos que representan el mismo concepto.

En el ejemplo que te estamos presentando, estos tres valores, el nombre, la edad y la cantidad de materias aprobadas, en su conjunto representan los datos de un alumno, es decir, en otras palabras, que **un alumno está formado por esos tres valores**.

Campos

Seguramente te estarás preguntando que, si bien los tres elementos están asociados a un mismo nombre lógico, debe haber alguna manera de poder acceder a cada dato. Esa manera es utilizando el nombre de cada elemento. **Cada elemento de la Estructura se denomina "campo"**.

En nuestro ejemplo, la estructura **Alumno** tiene tres campos. Y podrás acceder a cada campo por medio del operador punto "**.**" y el nombre del campo.

```
1   int main() {
2       Alumno alum;
3       cout << “Ingrese la edad del alumno”;
4       cin >> alum.Edad;  //Se escribe en el campo edad
5
6       cout << alum.Edad;  //Se lee el campo edad
7   }
```

- alum -> Nombre de Registro
- "." -> Operador punto
- Edad-> Nombre del Campo

Para acceder a la **Edad** debemos indicarlo con el nombre de la variable alum seguido del operador punto “.” y luego el nombre del campo al que queremos acceder, en este caso Edad.

Del mismo modo, si queremos asignar el nombre “Juan” como nombre del alumno debemos hacerlo de siguiente manera:

```
    alum.Nombre = “Juan”;
```

Algoritmos básicos

Una de las ventajas de utilizar estructuras es que se pueden agrupar en una misma variable todos los datos relacionados lógicamente. De esta manera, podemos manejar la abstracción para el ingreso y la salida de los datos.

Por ejemplo, si estamos trabajando con datos de alumnos, lo más simple sería pensar en una entidad **ALUMNO** como un único conjunto de datos (ocultando los datos que tiene un alumno).

Siguiendo con este concepto, podemos desarrollar los siguientes procedimientos.

```
1   Alumno obtenerAlumno() {
2       Alumno alum;
3       cout << "Ingrese nombre: ";
4       cin >> alum.Nombre;
5
6       cout << "Ingrese edad: ";
7       cin >> alum.Edad;
8
9       cout << “Ingrese materias aprobadas: ”;
10      cin >> alum.Aprobadas;
11
```

```
12      return alum;
13  }
14
15  void imprimir(Alumno alum) {
16      cout << "Nombre: " << alum.Nombre;
17      cout << "Edad: " << alum.Edad;
18      cout << "Aprobadas: " << alum.Aprobadas;
19  }
```

En la función **obtenerAlumno**, habrás notado que el tipo de retorno es **Alumno**. Esto nos permite simplificar y abstraer el tipo de dato, ya que estamos manejando un alumno como concepto de una entidad (sin importar los datos que tenga).

Por otro lado en el procedimiento imprimir, estamos pasando como parámetro por Valor una variable también del tipo Alumno, con lo cual logramos la misma abstracción.

Al utilizar la estructura y no los datos “sueltos” será mucho más simple el mantenimiento, ya que si querés agregar o eliminar datos al alumno, solo vas a tener que modificar la estructura del registro. Las especificaciones de las funciones y procedimientos que utilizan estas estructuras no se ven modificadas ya que siempre se estará pasando como parámetro un registro de ese tipo.

Diferencias con Arreglos

La principal diferencia que tienen las estructuras de datos o registros respecto de los arreglos es la **posibilidad de almacenar datos de distintos tipos bajo el mismo nombre** de variable. Recordemos que los arreglos, tanto las matrices como **los vectores, siempre son de un único tipo** de datos. Por lo cual si deseamos tener en un mismo lugar el nombre, legajo y las notas de los alumnos, no lo podríamos hacer con un único arreglo.

Podríamos recurrir a una matriz como la que hemos usado anteriormente y además otro vector con los nombres ya que son de otro tipo de dato.

int mat[N][4] y **string nombres[N]**

Y hacer la correspondencia por números de posición. Lo que daría gráficamente algo como lo siguiente:

	Nombre
0	Maria
1	Pepe
2	Julia
N	Carlos

	0 (Legajo)	1 (Nota 1)	2 (Nota 2)	3 (Nota 3)
0	12323	6	3	8
1	22112	4	8	7
2	33223	6	2	9
N	42322	7	2	10

Lo cual nos obligaría a estar accediendo y manteniendo ambas estructuras por separado, lo cual no es una abstracción muy potente.

En cambio con las estructuras generamos nuevos tipos de datos, que pueden combinar cualquier otro tipo de dato, los primitivos, los arreglos, y los nuevos que nosotros creemos. Por lo cual podemos realizar un vector de estructura del siguiente tipo:

Alumno alumnos[N]

Y entonces nuestras funciones de carga e impresión de datos ya no trabajan con datos “sueltos” sino con estructuras que representan abstracciones mejor pensadas:

```
1   void cargarAlumnos(Alumno alumnos[], int cant) {
2       for(int i = 0; i < cant; i++) {
3           alumnos[i] = obtenerAlumno();
4       }
5   }
6
7   void imprimir(Alumno alumnos[], int cant) {
8       for(int i = 0; i < cant; i++) {
9           imprimirAlumno(alumnos[i]);
10      }
11  }
```

Además de esta forma es bastante más específico el dato que se guarda en cada lugar, y no por posiciones, como lo hacemos en las matrices o vectores.

Una posible estructura para poder guardar todos los datos del alumno y sus notas sería:

```
1   struct Alumno {
2       int Legajo;
3       string Nombre;
4       int Nota1;
5       int Nota2;
6       int Nota3;
7   };
```

Y podemos reescribir nuestros algoritmos para que hagan uso de esta estructura en lugar de una matriz de enteros.

Unidad Temática N°8

~ Algoritmos de Ordenamiento y Búsqueda ~

“Dos peligros amenazan constantemente al mundo: orden y desorden.”

~ Paul Valéry

Búsqueda

A menudo nos veremos en la necesidad de encontrar un valor en un arreglo. Esto puede ser utilizado tanto para saber si algún valor ya está presente o recuperar algún registro para trabajar o imprimir sus datos.

Veremos 3 métodos de búsqueda distintos, cada uno tiene sus particularidades y en algunos casos algunas precondiciones que debe cumplir el arreglo para poder usarse.

Búsqueda Secuencial

Este algoritmo de búsqueda es el más sencillo de comprender, pero al mismo tiempo es el menos eficiente. Se reduce simplemente a recorrer el vector hasta encontrar el valor buscado.

Algoritmo

Como dijimos, recorremos el vector y por cada elemento evaluamos si es igual al valor buscado. Cabe destacar que lo que se devuelve no será el valor (que ya lo teníamos) sino la posición. En caso que no se encuentre el valor en el vector, devolveremos -1 :

```
1   int buscarSecuencial(int vec[], int, cant, int valor) {
2       for(int i = 0; i < cant; i++) {
3           if (vec[i] == valor) {
4               return i;
5           }
6       }
7
8       //Si no salio todavia, es que no estaba en el vector
9       return -1;
    }
```

Si bien es un algoritmo muy sencillo, como dijimos anteriormente es el más costoso en términos de eficiencia, ya que si el valor que buscamos está muy cerca del final, habremos recorrido casi la totalidad de los elementos.

Búsqueda por Posición Única Predecible (PUP)

Existen algunos casos donde podremos hacer una búsqueda directa apoyándonos en alguna clave que pueda ser equivalente a la posición de un elemento dentro del vector.

Supongamos que tenemos los registros de venta diaria de un mes, formados por los siguientes datos:

- Día
- Venta Diaria

Como nosotros sabemos que cantidad de días hay como máximo (31) y los días son consecutivos, entonces podemos hacer una equivalencia entre el número del día en el mes y la posición que ocupará la venta en el vector. Por lo tanto podemos utilizar un vector de **float** donde cada posición guardará la venta del día que le corresponde:

```
1   int main() {
2       float ventas[31];
3
4       for(int i = 0; i < 31; i++) {
5           cout << “Ingrese la venta del día “ << i + 1 << “: “;
6           cin >> ventas[i];
7       }
8   }
```

En este ejemplo, le pediremos al usuario que ingrese la venta de un día determinado, y si se mira con atención veremos que la equivalencia que estamos encontrando es:

posición = **día - 1**

Es decir la venta del día 4 estará en el vector en la posición 3, la del día 1 en la posición 0 y así sucesivamente.

Ahora supongamos que en realidad el usuario no tiene las ventas ya sumarizadas por día, sino que tiene un registro de venta por cada venta realizada:

- Día
- Valor venta

Si queremos poder realizar la carga de los datos para luego imprimir las ventas diarias y determinar cuál es el día de mayor venta, haremos uso y encontraremos el mayor beneficio del vector con PUP:

```
1   int main() {
2       float ventas[31];
3       int dia;
4       float vta;
5       cout << “Ingrese las ventas del mes” << endl;
6       for(int i = 0; i < 31; i++) {
7           cout << “Día: “;
8           cin >> dia;
9           cout << “Venta: ”;
10          cin >> vta;
11          ventas[dia - 1] = ventas[dia - 1] + vta; // Acumulo la venta leida
12      }
13  }
```

Vemos que durante la carga utilizaremos el día como clave para ubicarnos en el vector en la posición que le corresponda a ese día.

De esta forma tendremos el vector cargado con los acumulados de venta diario, sin necesidad de haber buscado por cada venta la posición en el vector de forma secuencial.

Como vemos existen algunas precondiciones para poder hacer uso de la búsqueda por PUP:

- Equivalencia entre alguna clave del dato y la posición: debe haber alguna forma de convertir una clave en una posición. La fórmula para convertir las claves a posiciones es siempre la misma:

  posicion = clave - valor_inicio_rango

- Se debe definir un rango conocido de claves: Los valores posibles de las claves deben ser conocidos de antemano
- El rango debe estar completo: para que no haya agujeros en el vector el rango de las claves debe ser completo, deben estar todos los valores.

Ejemplo: además del día (que vemos que cumple con las tres condiciones) pensemos algunos ejemplos más:

- Horas del día: Sabemos que son números del 0 al 23
- Meses del año: Rango del 1 al 12
- Legajos: Si sabemos que para alguna carga se cargan los datos de alumnos con legajos consecutivos del 50 al 150.

En el caso de que el rango no esté completo, igualmente podremos usar PUP pero siempre teniendo en cuenta que no podemos asegurar que los valores sean relevantes en todas las posiciones.

Búsqueda binaria

Si tuviéramos un vector ordenado, en el que no podemos establecer PUP, no tendríamos otra forma de encontrar cualquier elemento que haciendo una búsqueda secuencial. Pero si lo analizamos un poco, el hecho de que el vector ya esté ordenado debería ayudarnos a reducir el tiempo de búsqueda porque sabemos que cuanto más grande, más cerca del final estará (siempre que esté ordenado en forma ascendente). Para esto utilizaremos otro algoritmo que es la búsqueda binaria.

La búsqueda binaria es un algoritmo de búsqueda rápida con una complejidad de tiempo de ejecución de (log n). Este algoritmo de búsqueda funciona sobre el principio de dividir y conquistar. Para que este algoritmo funcione correctamente, el vector debe estar ordenado.

La búsqueda binaria busca un elemento en particular comparando el elemento más central del array. Si se produce una coincidencia, se devuelve el índice del elemento. **Si el elemento intermedio es mayor** que el elemento buscado, entonces el elemento se busca en el **sub-vector a la izquierda** del elemento central. **De lo contrario, el elemento se busca en el sub-vector a la derecha** del elemento central. Este proceso también continúa en el sub-vector hasta que el tamaño del sub-vector se reduce a cero.

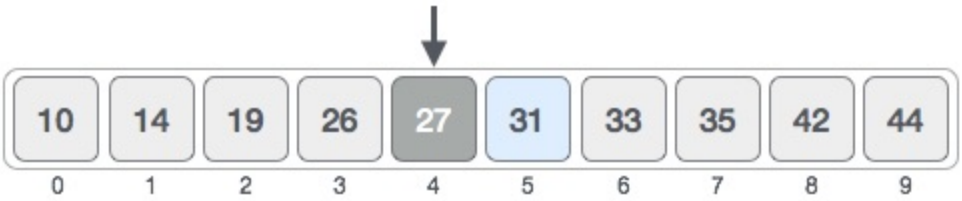
¿Cómo funciona la búsqueda binaria?

Para que una búsqueda binaria funcione, es obligatorio que el vector de datos se encuentre ordenado. Veamos un ejemplo gráfico. El siguiente es nuestro vector ordenado y supongamos que necesitamos buscar la ubicación del valor **31** utilizando la búsqueda binaria.

Primero, determinaremos la mitad del vector usando esta fórmula:

```
medio = inicio + (fin - inicio) / 2
```

Aquí resulta, 0 + (9 - 0) / 2 = 4 (valor entero de 4.5). Entonces, **4** es la mitad del vector.



Ahora **comparamos el valor almacenado en la posición 4 con el valor que se está buscando**, es decir, 31. Encontramos que el valor en la ubicación 4 es 27, que no es el buscado. **Como el valor que buscamos es mayor que 27 y tenemos un vector ordenado, ya sabemos que el valor objetivo debe estar en la parte superior del vector.** Es decir, podemos desestimar todo lo que está a la izquierda de la posición 4 que acabamos de evaluar.

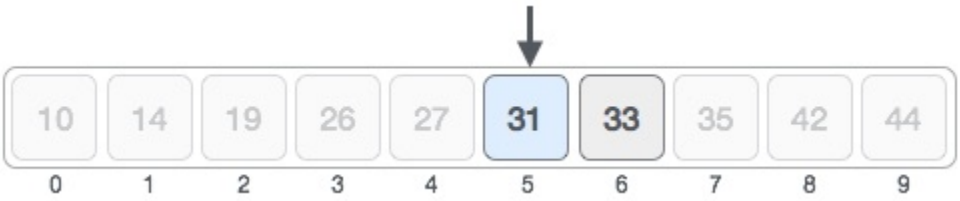
Cambiamos nuestro valor de **inicio** a **medio + 1** y volvemos a encontrar el nuevo valor medio.

```
inicio = medio + 1
medio = inicio + (fin - inicio) / 2
```

Nuestro nuevo medio ahora es 7. Comparamos el valor almacenado en la ubicación 7 con nuestro valor objetivo 31.

El valor almacenado en la ubicación 7 no coincide, es más que lo que estamos buscando. Por lo tanto, el valor debe estar en la parte inferior de esta ubicación.

Por lo tanto, calculamos el medio de nuevo. Esta vez son las 5.



Comparamos el valor almacenado en la ubicación 5 con nuestro valor objetivo. Encontramos que es un partido.

Concluimos que el valor objetivo 31 se almacena en la ubicación 5.

La clave está en que la búsqueda binaria divide a la mitad los elementos de búsqueda y, por lo tanto, reduce el recuento de comparaciones que se realizarán a números muy inferiores comparado con la búsqueda secuencial.

Veamos el algoritmo en C/C++

```
1   int busquedaBinaria(int arr[], int n, int x) {
2       // Cuando arranco evalúo todo el vector de 0 a n - 1
3       int inicio = 0;
4       int fin = n -1;
5
6       while (fin >= inicio) {
7           int mitad = inicio + (fin - inicio) / 2;
8
9           // Si el elemento es el del medio, devolvemos la posicion
10          if (arr[mitad] == x)
11              return mitad ;
12
13          // Si el elemento es menor entonces solo puede estar en la primer mitad
14          if (arr[mitad] > x) {
15              fin = mitad - 1; //Cambio el limite superior
16          } else {
17              inicio = mitad + 1;  // Cambio el limite inferior
18          }
19      }
20
21      // Si llegamos hasta aca es que el elemento no estaba
22      return -1;
23  }
```

Algoritmos de Ordenamiento

Como hemos visto en el caso de la búsqueda binaria, tener el vector ordenado nos permite realizar algunos algoritmos que de otra forma serían mucho más trabajosos. Además de que puede ser un requerimiento emitir algún listado ordenado.

Para poder ordenar un vector existen infinidad de algoritmos, los más eficientes intentarán realizar toda la operatoria sobre el mismo vector, dado que la memoria asignada de forma

estática queda asignada hasta que el programa finalice, y no es eficiente tener duplicado el vector simplemente como estructura auxiliar para el ordenamiento.

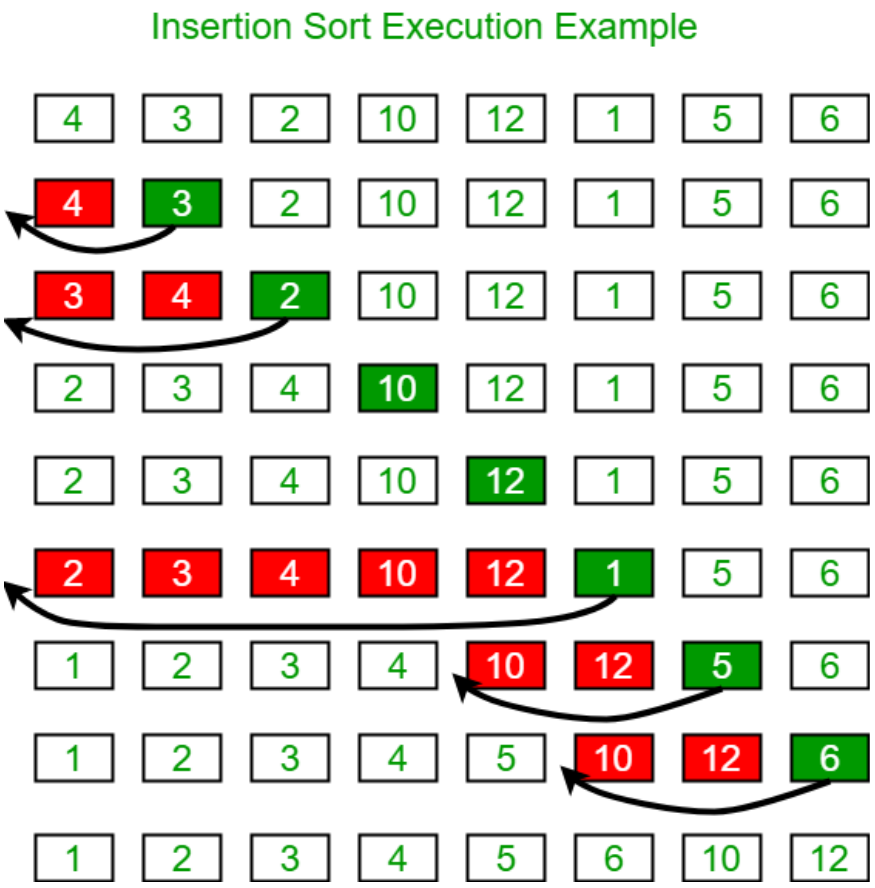
Nosotros veremos solamente dos algoritmos, siendo el más eficiente Burbujeo Mejorado.

Insertion Sort

Este algoritmo es uno de los más sencillos (y por lo tanto no tan eficiente!) consiste en ordenar los elementos similar a como hacemos con los naipes, vamos insertando el valor en donde le corresponde mirando los naipes que tengo hasta el momento.

Veamos un ejemplo gráfico:

Como vemos lo que hace es ubicar cada elemento en donde le corresponde en el subvector a la izquierda del elemento.



Veamos otro ejemplo paso a paso:

12, 11, 13, 5, 6

Iteramos para i = 1 (arrancamos desde el 2do elemento) hasta 4 (último elemento del array)

i = 1. Como 11 es más chico que 12, mueve 12 e inserta el 11 en su lugar

11, 12, 13, 5, 6

i = 2. 13 queda en su posición porque todos los elementos de A[0..I-1] son más chicos

11, 12, 13, 5, 6

i = 3. El 5 se mueve al principio de todo y todos los otros elementos del 11 al 13 se desplazan un lugar a la derecha.

5, 11, 12, 13, 6

i = 4. 6 se moverá a la posición después del 5 y los elementos del 11 al 13 se mueven un lugar.

5, 6, 11, 12, 13

Cuidado, si bien pareciera que solo necesitamos recorrer el vector una vez, en realidad cada vez que tuvimos que desplazar elementos, tuvimos que recorrer un subvector y esto varias veces, por eso decimos que no es el más eficiente de los algoritmos.

Veamos el código

```
1   void insertionSort(int arr[], int n) {
2      int i, key, j;
3      for (i = 1; i < n; i++)
4      {
5          key = arr[i];
6          j = i - 1;
7
8          /* Movemos los elementos de arr[0..i-1], que son
9          mas grandes que key, a la posicion siguiente
10         a su posicion actual */
11         while (j >= 0 && arr[j] > key)
12         {
13             arr[j + 1] = arr[j];
14             j = j - 1;
15         }
16         arr[j + 1] = key;
17     }
18 }
```

Burbujeo

Este método de ordenamiento consiste en comparar cada par de valores adyacentes, e intercambiarlos en caso que no estén en el orden buscado

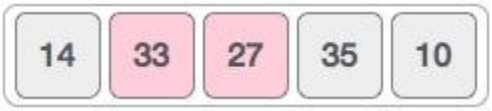
¿Cómo funciona?

Tomemos un vector desordenado para nuestro ejemplo. El método de burbujeo precisará O(n2) repeticiones así que lo mantendremos lo más sencillo posible

Burbujeo comienza con los primeros dos elementos, comparándolos y verificando cuál es más grande.

En este caso el valor 33 es más grande que 14, por lo cual ya está ordenado. Luego nos movemos un lugar y compararemos el 33 con el 27.

Y descubrimos que el 27 es más chico que el 33, por lo cual estos dos valores deben ser intercambiados.



El nuevo vector debería quedar de la siguiente forma:

Luego nos volvemos a mover un lugar y comparamos el 33 con el 35. Que al ya estar ordenados, quedan donde están.

Luego evaluamos el 35 con el 10

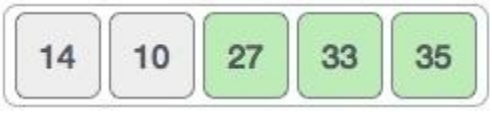
Como el 10 es más chico necesitamos hacer el intercambio ya que no están en orden:

Así llegamos al final del vector, al final de esta primer iteración el vector debería lucir así

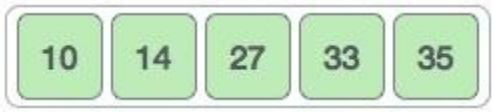
Como decíamos, vemos que al final de la primer iteración, el valor 35 ya encontró su lugar definitivo en el vector ordenado. Es decir en la primer iteración ya logramos tener un elemento ordenado. Por lo tanto ahora al hacer la segunda pasada, no necesitaremos llegar hasta ese valor para evaluar, sino que deberemos evaluar los valores hasta N - 2

Veamos como va quedando el vector al final de cada iteración en las siguientes pasadas. Luego de la segunda iteración

Vemos que ahora ya son dos elementos los ordenados, el 33 y el 35 ya están en donde quedarán finalmente. En la tercer iteración quedarán el 27, 33, 35 ya ordenados:



Luego el vector quedará finalmente ordenado:



Veamos el código para realizar el algoritmo en su versión más sencilla (y menos eficiente)

```
1   void burbujeo(int arr[], int n)
2   {
3      int i, j, aux;
4      for (i = 0; i < n-1; i++) {
5
6          // Los ultimos i elementos ya estan ordenados
7          for (j = 0; j < n-i-1; j++) {
8              if (arr[j] > arr[j+1]) {
9                  aux = arr[j];
10                 arr[j] = arr[j+1];
11                 arr[j+1] = aux;
12             }
13         }
14     }
15  }
```

El inconveniente de este algoritmo es que si a la segunda pasada, el vector ya está completamente ordenado, nosotros **igualmente hacemos n- 1 iteraciones**. Para evitar esto lo que haremos es agregar una bandera o flag, que nos permita saber si pudimos completar una pasada completa sin hacer intercambios de valores**, es decir si el vector ya está ordenado**.

Burbujeo Mejorado

Como dijimos, mejoramos el burbujeo agregando el flag ordenado, al iniciar cada pasada lo ponemos en **true** si en esa pasada hubo al menos un valor que no estaba en orden, se cambia a **false.**

```
1   void burbuejoMejor(int arr[], int n)
2   {
3      int i, j, aux;
4      int i = 0;
5      bool ordenado = false;
6      while (i < n && !ordenado) {
7         ordenado = true;  // Arranco asumiendo que si esta ordenado
8         // Los ultimos i elementos ya estan ordenados
9         for (j = 0; j < n-i-1; j++) {
10            if (arr[j] > arr[j+1]) {
                  aux = arr[j];
                  arr[j] = arr[j+1];
                  arr[j+1] = aux;
                  ordenado = false; // Cambio el flag si hice un swap
              }
          }
          i++;
      }
   }
```

Unidad Temática N°9

~ Apareo y Corte de Control~

"La función de un buen software es hacer que lo complejo aparente ser simple."

~ Grady Booch

Apareo de Vectores

El apareo de vectores (o mergeo) nos permite combinar dos vectores en un tercer vector que contendrá elementos de los vectores originales. En su versión más sencilla el vector resultante contendrá todos los elementos de ambos vectores, pero es muy sencillo adaptar el algoritmo original para agregar nuevas reglas o controles al formar el vector resultante.

Vamos de a poco, ¿Qué queremos lograr?

El objetivo de mínima es dados dos vectores obtener todos los elementos de ambos en un único vector. Ejemplo:

A = {-5, -3, 0, 1, 4, 6, 7, 9, 14} B = {-7, -5, -4, -1, 2, 5, 8, 10, 16}

Si alguien nos pidiera un vector C “con todos los elementos de A y B” sin mayores aclaraciones, podríamos decir que C = {-5, -3, 0, 1, 4, 6, 7, 9, 14, -7, -5, -4, -1, 2, 5, 8, 10, 16} cumple lo pedido. El vector es el resultado de “anexar” todos los elementos de B luego de los de A en un vector C.

Apareo de vectores ordenados

¿Pero qué sucedería si nos aclaran que A y B se encuentran ordenados y el objetivo es obtener un vector C ordenado de igual forma?

Es decir dados los mismos A y B el resultado debería ser:

C = {-7, -5, -5, -4, -3, -1, 0, 1, 2, 4, 5, 6, 7, 8, 9, 10, 14, 16}

Nuevamente hay una forma sencilla de resolverlo, primero “anexar” y luego ordenar el vector resultante (utilizando burbujeo por ejemplo). El problema es que si realizamos estos pasos estaremos recorriendo ambos vectores una vez, y posteriormente pagaremos el costo de ordenar el vector resultante, un costo que dependiendo del tamaño del vector podría ser imposible de justificar.

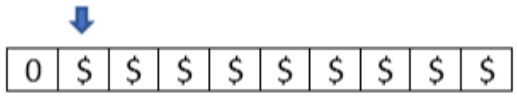
El dato de que ambos vectores están ordenados es el que nos ayudará a resolver este problema con una sola recorrida de ambos vectores.

Dado que el objetivo es obtener el vector resultante ordenado en forma ascendente (o descendente) igual que como están los vectores originales, podemos decir que lo que deseamos es llenar el vector tomando los valores más chicos. Y al estar ambos vectores ordenados,

sabemos que los valores más chicos se encuentran al inicio. Por lo cual lo que haremos será comparar valor por valor de ambos vectores iniciando desde la posición 0, y colocaremos en el vector resultante el valor que sea menor (o mayor si el orden fuera descendente).

Al colocar el valor menor en el vector resultante, en el vector original ya no nos interesa más, con lo cual pasaremos al siguiente elemento, del vector que hayamos tomado el valor. Repitiendo estas simples acciones completamos el vector resultante. Veámoslo con un gráfico:

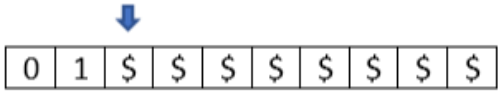
Nos paramos al inicio de ambos vectores y comparamos sus valores, como resultado obtenemos que 0 es el menor valor y es el primer elemento de nuestro vector resultante:



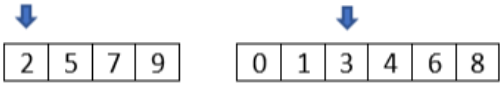
Colocamos el 0, y nos movemos para que el próximo valor se guarde en la posición siguiente.

Como ya tomamos el primer elemento del vector B, nos movemos para evaluar el siguiente, con lo cual ahora comparamos el primer elemento del vector A y el segundo de B

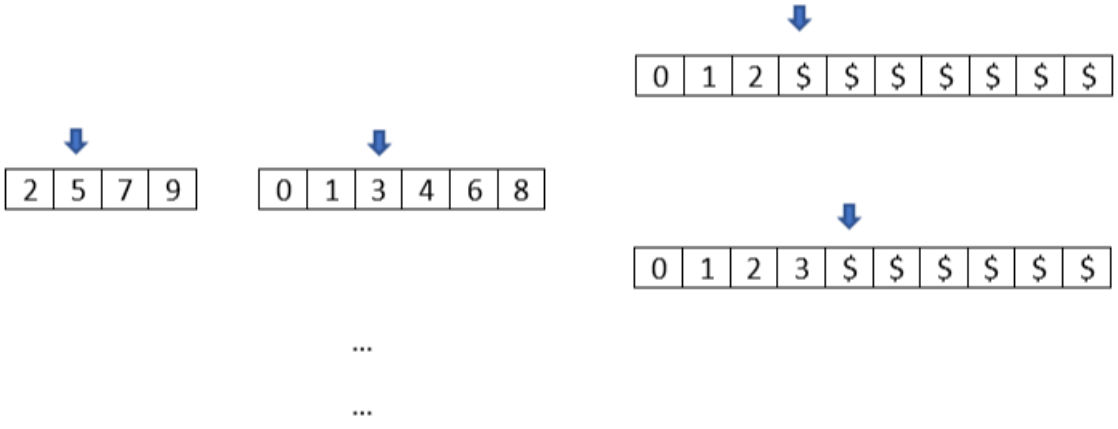
Nuevamente el elemento lo tomamos de B ya que 1 es menor que 2



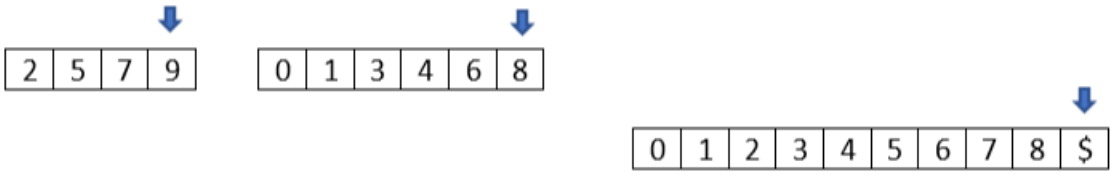
Nuevamente comparamos los elementos en donde estamos parados en cada uno de los vectores:



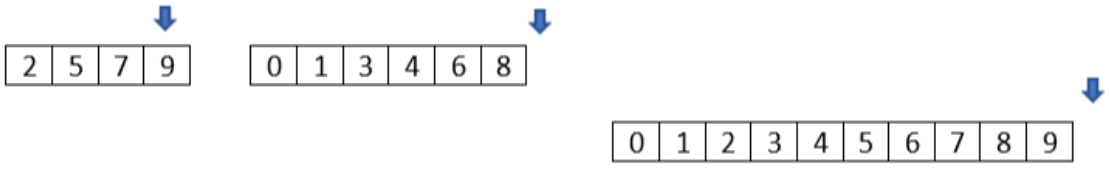
En este caso tomaremos el valor de A ya que 2 es menor que 3 y por lo tanto moveremos su posición para la próxima evaluación.



La misma operatoria se repetirá hasta que ya no pueda comparar elementos, es decir cuando alcancemos el final de algunos de los dos vectores. En nuestro ejemplo sucederá al llegar al final del vector B.



Como ya no hay elementos de B, lo único que queda es pasar todos los elementos de A.



Si en lugar de acabarse B se hubiera acabado A se haría lo mismo, se pasarían todos los elementos restantes de B.

Como desafío podés intentar hacer la implementación en código del algoritmo que acabamos de hacer

El código para hacer el apareo de dos vectores ordenado es el siguiente:

```
1   void apareo(int vecA[], int n, int vecB[], int m, int vecC[], int &k) {
2      // Contadores para la posicion de los vectores A y B.
3      int i = 0, j = 0;
4      // Contador para posicionarse en el vector resultante.
5      k = 0;
6
7       // Mientras pueda comparar valores (al menos uno de los vectores tiene
8   valor)
9      while (i < n && j < m) {
10        // Comparo los valores de los vectores
11        if (vecA[i] < vecB[j]) {
12           // Coloco el elemento de A porque es menor
13           vecC[k] = vecA[i];
14           // Me muevo en el vector A
15           i++;
16        } else {
17           vecC[k] = vecB[j];
18           j++;
19        }
20        // Incremento el contador de la posicion del vector resultante
21        k++;
22     }
23
24     // Paso todos los elementos restantes de A
25     while(i < n) {
26        vecC[k] = vecA[i];
27        i++;
28        k++;
29     }
30
31     // Paso todos los elementos restantes de B
32     while(j < m) {
33        vecC[k] = vecB[j];
34        j++;
35        k++;
36     }
   }
```

Como dijimos anteriormente, esta es la versión más sencilla pero sobre el mismo algoritmo podrían agregarse distintas validaciones como no poner valores repetidos, sólo porner valores repetidos, sólo poner valores distintos, etc.

Tener en cuenta, este algoritmo de apareo sirve solamente para aparear vectores ordenados en forma ascendente, es decir ambos vectores deben estar ordenados en forma ascendente y el resultante también quedará ordenado de esa forma.

Apareo de vectores de registros

Al igual que en los algoritmos vistos anteriormente, lo único que deberemos hacer para poder usar este mismo algoritmo con estructuras propias, será modificar los tipos de datos y la condición de comparación. De más está decir que los vectores deberán estar ordenados por el mismo criterio que usemos para la comparación en nuestro algoritmo.

```
1   struct Alumno {
2      int legajo;
3      int nota1;
4      int nota2;
5   };
6
7
8   void apareo(Alumno  vecA[], int n, Alumno vecB[], int m, Alumno vecC[], int &k)
9   {
10     // Contadores para la posicion de los vectores A y B.
11     int i = 0, j = 0;
12     // Contador para posicionarse en el vector resultante.
13     k = 0;
14
15      // Mientras pueda comparar valores (al menos uno de los vectores tiene
16  valor)
17     while (i < n && j < m) {
18        // Comparo los valores de los vectores
19        if (vecA[i].legajo < vecB[j].legajo) {
20           // Coloco el elemento de A porque es menor
21           vecC[k] = vecA[i];
22           // Me muevo en el vector A
23           i++;
24        } else {
25           vecC[k] = vecB[j];
26           j++;
27        }
28        // Incremento el contador de la posicion del vector resultante
29        k++;
30     }
31
32     // Paso todos los elementos restantes de A
33     while(i < n) {
34        vecC[k] = vecA[i];
35        i++;
36        k++;
37     }
38
39     // Paso todos los elementos restantes de B
40     while(j < m) {
41        vecC[k] = vecB[j];
42        j++;
43        k++;
44     }
   }
```

Corte de Control

El corte de control es una técnica que nos permitirá obtener información a partir de un conjunto de datos. Su uso principalmente es obtener impresiones, cantidades, promedios, etc. agrupados con algún criterio, y parte de la base de que los datos originales cumplen o se puede hacer que cumplan con el agrupamiento necesario para su procesamiento.

¿Qué quiere decir todo esto? Mejor veámoslo con un ejemplo

Supongamos que tenemos un conjunto de datos ya cargado con la información del presentismo de cada alumno de la materia:

Legajo	Fecha	Presente
11709099	20200504	Verdadero
13423223	20200504	Falso
12936274	20200518	Falso
11709099	20200525	Verdadero

Si alguien nos preguntara cuántas faltas tiene el alumno con legajo 11709099 una forma sería recorrer todo el conjunto y preguntar si es el legajo buscado, en caso de serlo contar en caso que Presente sea Falso. El problema, una vez más, es la cantidad de recorridas que deberíamos hacer si necesitamos imprimir el siguiente listado

Legajo	Faltas
11709099	2
...	...

Deberíamos recorrer el conjunto completo por cada Legajo distinto que haya, y para complicar más la tarea, al no saber cuántos legajos son realmente no podremos utilizar una estructura auxiliar como un vector de forma óptima ya que estaremos obligados a declararlo de una cantidad excesiva por las dudas.

Utilizar corte de control nos permitirá obtener el listado deseado pero en una única recorrida

Pero cuidado, es **requisito indispensable** que los datos que contiene el archivo estén agrupados y ordenados por la condición de Corte de Control que se solicita, en este caso por legajo.

Una característica de los datos, para aplicar la técnica de corte de control, es que se repitan, en posiciones consecutivas, valores de aquellos campos sobre los que se pretende realizar el corte, en este caso el alumno o legajo más propiamente dicho.

Siguiendo con el ejemplo, dado que para un mismo legajo existen muchos registros de presentismo, deberemos evaluar mientras que el legajo no cambie el valor del presente y contar en caso que corresponda.

Veamos como quedaría nuestro algoritmo que imprime el listado deseado.

```
1   struct Presentismo {
2      int legajo;
3      int fecha;
4      bool presente;
5   };
6
7   void listarPresentismo(Presentismo vec[], int n) {
8      int i = 0;
9      int ausentes = 0;
10     int key;
11
12     // Inicializamos contadores, acumuladores, etc. generales
13
14     // El primer ciclo es el que recorre el lote completo
15     while(i < n) {
16
17        // Guardo el valor de la clave o agrupador
18        key = vec[i].legajo;
19
20        // Inicializo contadores, acumuladores, etc cada sublote
21        ausentes = 0;
22
23        // El segundo ciclo se mantiene por el sublote, mientras sea el mismo
24        // legajo y aun no se haya acabado el vector
25        while(i < n && key == vec[i].legajo) {
26           // Cuento si es un registro de ausente
27           if (!vec[i].presente) {
28              ausentes++;
29           }
30
31           i++;   // Avanza a la siguiente posicion
32        }
33
34        // Mostramos resultados por cada sublote (legajo)
35        cout << “Legajo: ” << key << “ faltas: ” << ausentes << endl;
36     }
37
38     // Mostramos resultados generales
39  }
```

Corte de Control por múltiples criterios

En el ejemplo anterior nuestro criterio de corte es el legajo dado que el listado era por cada materia, pero qué pasaría si en realidad la tecnicatura tuviera un único listado como el siguiente:

Materia	Legajo	Fecha	Presente
INGLES1	11709099	20200529	Verdadero
PROG1	11709099	20200521	Verdadero
PROG1	11709099	20200528	Falso
PROG2	1232342	20200517	Verdadero

Si el listado resultante ahora es las faltas por materia por alumno:

Materia	Legajo	Faltas
INGLES1	11709099	0
PROG1	11709099	1

Deberemos agregar un nuevo ciclo al algoritmo anterior para generar un nuevo sublote anidado, es decir nuestro primer nivel será el sublote de la materia, y el segundo el legajo.

Cabe remarcar que siempre para poder utilizar corte de control el conjunto de datos debe estar ordenado (o poder ordenarse) por los criterios que se necesiten utilizar en el corte de control.

¿Cuándo conviene usarlo?

Para poder responder la pregunta de si es necesario o útil utilizar corte de control para resolver un problema deberemos analizar los datos brindados y el listado o conjunto de datos al que queremos llegar. En el ejemplo anterior no hay otra forma de resolverlo más que el corte de control, dado que no sabemos la cantidad de alumnos total, tampoco las materias, y menos la cantidad de clases dictadas por cada una. En general el corte de control nos será de gran utilidad cuando no hay otra forma de realizar los cálculos por la imposibilidad de almacenar los contadores, acumuladores, etc. que precisaríamos para poder resolverlo.

Unidad Temática N°10

~ Depuración ~

“Depurar es como ser el detective en una película de crimen en la que tú también eres el asesino.”

~ Filipe Fortes

Depuración

Definiciones

Bug

Es un defecto en el código de un programa que desencadena en una falla del producto.

Falla

Es la manifestación de un defecto en el código.

Depuradores

Son programas que permiten, como su nombre lo indica, depurar el código fuente. Muchas veces incluyen interfaces gráficas y se encuentran integrados en un entorno de desarrollo. Proveen herramientas que permiten:

- Recorrer paso a paso un programa.
- Detenerse en una línea particular del código, esto se denomina breakpoint o punto de interrupción. Los mismos pueden ser condicionales.
- Inspeccionar el valor de las variables locales y globales
- Observar la secuencia de ejecución del programa (stack trace)

La prueba a ciegas con un depurador no suele ser productiva. Es más útil usarla para descubrir el estado del programa cuando falla, y luego pensar sobre cómo pudo haber ocurrido dicha falla.

Errores fáciles de corregir

La depuración implica razonamiento en retrospectiva. Se debe empezar el análisis desde el punto donde surgió la falla e ir hacia atrás para descubrir las causas. Cuando se tenga una explicación completa, sabrá que arreglar.

Busque patrones familiares

Analice si el error no lo vió antes. Los errores comunes tienen síntomas característicos.

- Errores de conversión de tipos
- Acceso a memoria fuera de los límites
- No inicializar variables

Muchos de estos errores pueden ser detectados por el compilador si se habilitan las verificaciones correspondientes.

Examine los cambios más recientes

Una práctica recomendada es cambiar una cosa a la vez. Si sigue esta regla, es muy probable que el error esté en el nuevo código o ha sido puesto en evidencia por lo último que haya hecho.

Una revisión cuidadosa ayuda a localizar el problema.

En este punto son de gran ayuda los sistemas de control de código (git, svn, etc.)

No cometa dos veces el mismo error.

Después de haber encontrado y corregido un error, revise el código para ver si el mismo error se encuentra en otra parte del código.

Corríjalo ahora, no después

Cuando probando su programa detecte un error, aunque no sea fatal, corríjalo en el momento en que lo detectó, dejarlo para después puede ser más crítico y costoso de corregir.

Obtenga una reconstrucción de la pila de ejecución

Un uso muy común de los depuradores es que permiten hacer un análisis post-mortem. Para esto se obtiene y analiza la pila de ejecución del programa (*stack trace*) la cuál nos brinda la línea de código donde se produjo la falla, además también muestra todas las llamadas a funciones que se fueron sucediendo.

Ejemplo:

```
Program received signal SIGSEGV, Segmentation fault. At ejercicio3.cpp:62

[debug]#0  0x000000000040124c in ejercicio3::ejecutar () at ejercicio3.cpp:62
[debug]#1  0x0000000000401552 in main () at main.cpp:25
[debug]>>>>>>cb_gdb:
```

La pila de ejecución debe leerse desde abajo hacia arriba para seguir el flujo de ejecución del programa.

En este ejemplo vemos que:

- lo primero que se ejecutó es la instrucción que se encuentra en la función main() en la línea 25 del archivo main.cpp
- lo segundo que se ejecutó es la instrucción que se encuentra en la función ejecutar() en la línea 62 del archivo ejercicio3.cpp

El mensaje siguiente nos está indicando que se produjo un error en dicha línea.

Errores difíciles de corregir

Hay ocasiones cuando no se tiene la menor pista acerca del por qué de la falla detectada. A eso nos referimos con *errores difíciles de corregir*. Algunas recomendaciones al respecto son:

Trate de hacer que el error sea reproducible

El primer paso es asegurarse de que puede reproducir el error cuando lo necesite. Diseñe casos de prueba, parámetros y entradas, que provoquen la falla.

Esto ayudará a analizar el problema, encontrar su solución y verificar que no vuelva a ocurrir.

Divide y vencerás

Esta técnica apunta a acotar el fragmento del código donde se genera el error. Para esto suele ser útil saber cuál fue el flujo de ejecución del código, ya que esto nos permitirá descartar algunas partes del mismo. Este accionar se puede repetir hasta dar con la línea que genera el error.

Estudie la numerología de los errores

A veces un patrón en la numerología de los casos de prueba diseñados, que permiten reproducir la falla, ofrece una pista que nos permite enfocar la búsqueda. Analizar los datos de entrada que provocan la falla y su relación puede llevarnos directamente a detectar el error.

Despliegue salidas para situar la búsqueda del error

Cuando no se entiende lo que el programa está haciendo, agregar instrucciones para mostar más información puede ser una forma sencilla y efectiva de averiguarlo. Esta técnica permitirá mejorar su entendimiento sobre el flujo de ejecución del código.

Si va a imprimir el valor de una variable, déle siempre el mismo formato.

Si muestra punteros, hágalo utilizando el formato %x(hexadecimal) o %p(puntero). Esto le ayudará a ver si dos punteros están relacionados.

Escriba un archivo de registro (log)

Este punto es similar al anterior, sólo que enviando la salida a un archivo que contenga la información en un formato fijo. Cuando el programa termina de forma repentina, el registro contendrá la información hasta ese punto.

Asegúrese de vaciar los buffers de E/S para que los registros finales aparezcan en el archivo. En C, una llamada a fflush garantiza que toda la salida se escriba antes de que falle el programa.

Tipos de Errores

A modo de resumen y desde un punto de vista conceptual, podemos agrupar los errores en tres tipos: sintácticos, lógicos y de regresión.

Errores Sintácticos

Son aquellos que se producen cuando el programa viola la sintaxis, es decir las reglas de gramática del lenguaje.

Estos errores suelen ser detectados por el compilador y son fáciles de corregir. Algunos ejemplos de estos errores son:

- Omisión de punto y coma al final de una sentencia.
- Comentarios multilínea sin cerrar. Falta de la secuencia */
- Olvido de doble comillas para cerrar una cadena
- Etc.

Errores Lógicos

Este tipo de error, representa un error del programador en el diseño del algoritmo. Son, por lo general, difíciles de encontrar y aislar. Estos errores no son detectados por el compilador.

Algunos de estos errores puden producirse por:

- Falta de paréntesis en sentencias que calculan algo.
- Falta de inicialización de variables
- Falta de verificación en los datos de entrada.
- Etc.

Ejemplos:

```
// el error está en que el tercer * debería ser un +
double peso = densidad * 5.25 * PI * pow(longitud, 5)/4.0

// esta fórmula no es la correcta
grados_centigrados = grados_farenheit * temperatura_centigrados
```

Errores de Regresión

Son errores que se crean de forma accidental cuando se intenta corregir otro error.

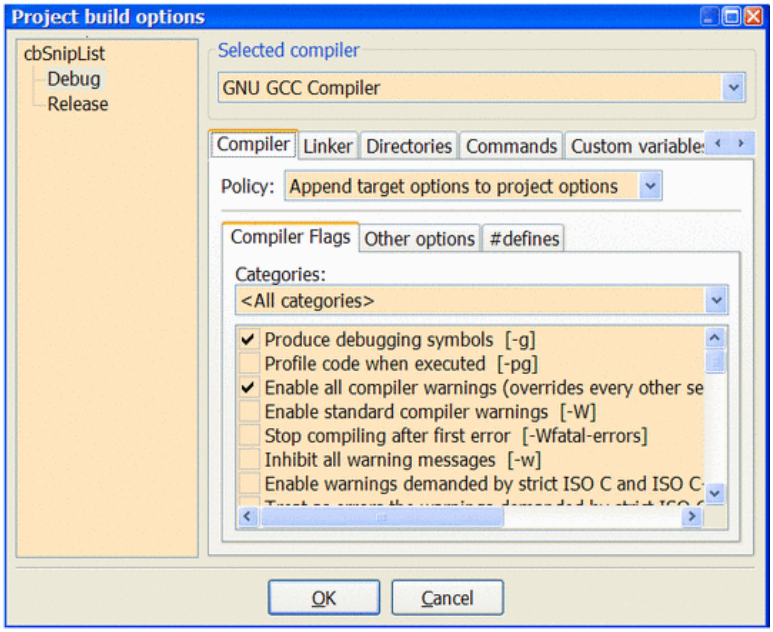
Como regla, y como ya mencionamos antes, antes de compruebe que el error no se produce en otra parte del código y que realmente fue corregido en su totalidad.

Depurando con CodeBlocks

Configurar CodeBlocks para depurar

El primer paso antes de poder depurar nuestro código es configurar el IDE para tal fin. Para ello debemos acceder al menú *Proyecto➔Opciones de Compilación (Project ➔ Build Options)*

Allí veremos una pantalla como la siguiente:



En el cuadro de la izquierda vemos el listado de modos de compilación, Debug (Depuración) y Release (Lanzamiento).

Debemos seleccionar la opción *Debug* y en el cuadro de la derecha vemos las opciones de compilación (Compiler Flags)

Opciones para construir una versión de depuración

Para poder generar una versión de depuración de su código, debe asegurarse que las siguientes opciones estén activas:

- Obligatorias
  - -g: compila el código fuente generando información de depuración que es utilizada por el depurador.
  - Opcionales pero útiles
    - -Wall: habilita todas las advertencias sobre construcciones comunes de error.

- -Wfatal-errors: frena el proceso de compilación ante el primer error
- Sólo Opcionales
  - -pedantic: habilita las advertencias exigidas por el estándar ISO
  - -pedantic-errors: trata como un error las advertencias exigidas por el estándar ISO

Compilar para depurar

Una vez activadas las opciones mencionadas, ya podemos generar nuestra versión de depuración.

Para ello, desde la ventana principal, accedemos al menú *Depurar* ➔ *Comenzar/Continuar.*

Recuerde que si hizo cambios o previamente había compilado en modo *Lanzamiento (Release)* deberá recompilar los fuentes.

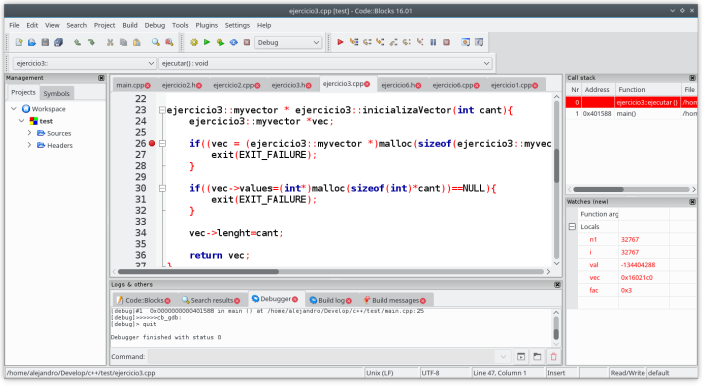
Establecer un Punto de Interrupción (breakpoint)

Un breakpoint o punto de interrupción es una marca que se establece sobre una línea del código para que al momento de depurar, el flujo de ejecución se detenga en la misma.

Para generar un punto de interrupción podemos:

- Acceder al menú *Depurar* ➔ *Establecer Punto de interrupción*
- O bien hacer click sobre el margen izquierdo (al lado del número de línea)

Una vez establecido el punto de interrupción, presionamos el botón *Depurar* y el programa se ejecutará hasta el primer punto de interrupción encontrado.



Observar el valor de una variable

Cuando estamos depurando el código, suele ser útil conocer el valor de las variables en un determinado momento. Para esto podemos agregar la ventana *Watches* la cuál nos mostrará las variables locales y globales y su valor en cada momento.

Para agregar la ventana *Watches*, debemos acceder al menu *Depuración➔ Ventanas de Depuración ➔ Watches*.

En la ventana *Watches*, también podemos evaluar expresiones que involucren 1 o más variables. Para esto, en la última fila de la ventan, hacemos doble click y escribimos la expresión.

Ver la pila de ejecución

Para agregar la ventana *Pila de ejecución*, debemos acceder al menu *Depuración➔ Ventanas de Depuración ➔ Pila de ejecución.*

Unidad Temática N°11

~ TAD ~

"Si ha elegido las estructuras de datos correctas y las cosas están bien organizadas, los algoritmos casi siempre serán evidentes.".

~ Rob Pike

Tipo Abstracto de Datos (TAD)

El rol de la abstracción

A lo largo de la historia, las personas han tenido que tratar con problemas complejos, problemas que involucran muchas variables, escenarios posibles, etc. Tratar de encontrar una solución a esos problemas prestando atención a cada detalle, realmente es una tarea compleja es por eso que surge la *abstracción* como la "herramienta" utilizada para tratar este tipo de problemas. La abstracción resulta ser la herramienta más eficaz para lidiar con problemas complejos ya que nos permite concentrarnos en los aspectos fundamentales del problema a resolver, dejando de lado los detalles no esenciales.

Definición	La abstracción es la capacidad para encapsular y aislar la información, de diseño y ejecución

Desde el surgimiento del software y los lenguajes de programación, podemos ver que la abstracción se repite una y otra vez hasta nuestros días.

Inicialmente la programación se realizaba a nivel de hardware, luego se comenzó a programar microprocesadores con instrucciones de *secuencia* y *salto*. Posteriormente, con el advenimiento de los lenguajes de alto nivel surgieron instrucciones que permiten controlar el flujo de ejecución con estructuras de control y repetición (if, while, for, etc), a esto se lo llama **abstracción de control** y también surgieron los tipos de datos, que permitió crear variables de un tipo de dato particular, este cambio permitió dar los primeros pasos en la **abstracción de datos**.

Un refinamiento posterior dentro de la abstracción de control es la **abstracción procedimental.** Esta técnica permite diseñar software de forma modular. Se basa en la utilización de procedimientos o funciones sin preocuparse de cómo están implementadas, es decir, sólo conociendo qué hacen, qué entrada requieren y qué salida tienen.

Tipo abstracto de dato

Con todo lo mencionado anteriormente, los programadores comenzaron a darse cuenta de que para crear software de calidad y confiable, la solución era modularizar el código.

Pero...¿Qué es un módulo?

Definición	Modularizar: es una técnica de construcción de software que permite separar los datos y procedimientos en una parte privada - sólo accesible dentro del módulo - y una parte pública - accesible por fuera del módulo -.

Entonces un módulo es una pieza de software diseñada con esta separación de datos y procedimientos públicos y privados presente desde el primer momento.

¿Y un tipo abstracto de dato?

Debemos recordar que un tipo de dato, en general, queda definido por un conjunto los valores (dominio) y las operaciones definidas sobre dichos valores.

Ahora bien, un tipo de dato abstracto o TAD es entonces un tipo de dato definido por el programador, que se puede utilizar de forma similar a los tipos de datos definidos por el sistema.

Por ejemplo, si definimos el tipo de dato *vectorEntero*, sus posibles valores serían números enteros y las operaciones podrían ser *cargarVector, ordenarVector, buscarEnVector*, etc.

La modularización se utiliza frecuentemente como una técnica para implementar los TAD ya que para poder construir un TAD, se debe poder:

- Exponer una definición del tipo (formal o informal).
- Hacer disponible un conjunto de operaciones que permitan operar sobre los datos.
- Proteger los datos de forma tal que sólo se pueda interactuar con ellos a través de las operaciones definidas.
- Poder generar múltiples instancias del tipo.

Ventajas de los TAD

La utilización de TAD para el desarrollo de software nos proporciona varias ventajas (respecto de no hacerlo), mencionamos algunas:

1. Mejora la conceptualización, comprensión y modelización del mundo real. Ayuda a clarificar los objetos (del mundo real) que intervienen en el problema y sus comportamientos.
2. Mejora la robustez del sistema. Nos permite realizar una verificación de tipos y evitar errores en tiempo de ejecución.
3. Separa la implementación de la especificación. Esto permite modificar la implementación (parte privada) sin cambiar la interfaz (parte pública) del tipo de dato.

4. Permite la extensibilidad del sistema. La utilización (y reutilización) de módulos permiten crear y mantener software con mayor facilidad
5. Agrupan y localizan las operaciones y la representación de atributos mejorando la semántica del tipo.

Un programa que utiliza un TAD, lo hace teniendo en cuenta las operaciones que tiene sin interesarse en la representación física de los datos ni en la implementación de las operaciones, es decir, utilizan el TAD a partir de su interfaz. Esto permite que se modifique la implementación sin alterar el funcionamiento de los programas que lo utilizan.

Especificación de un TAD

Como mencionamos previamente, un TAD está formado por un conjunto de datos y operaciones. La especificación de un TAD consta de 2 partes, la descripción del conjunto de datos y las operaciones definidas.

El objetivo es poder describir el comportamiento del TAD.

La especificación del TAD puede tener un enfoque informal, donde se describe tanto los datos como las operaciones con un lenguaje natural o bien puede tener un enfoque formal donde se debe suministrar un conjunto de axiomas que describen las operaciones sintáctica y semánticamente.

En este curso sólo veremos la especificación informal.

Especificación informal

El formato que generalmente se utiliza es especificar el nombre del TAD y los datos:

TAD <nombre del tipo> (<valores y su descripción>)

y a continuación cada una de las operaciones:

<nombreOperación>(<lista_de_argumentos>). <Descripción funcional>.

Ejemplo: TAD “Conjunto”

TAD Conjunto (colección de elementos sin duplicados, que pueden estar en cualquier orden, utilizado para representar los conjuntos matemáticos y sus operaciones).

Operaciones:

- conjuntoVacio. Crea un conjunto sin elementos.
- añadir(conjunto, elemento). Comprueba si el elemento pertenece al conjunto. En caso negativo lo añade.
- eliminar(conjunto, elemento). Comprueba si el elemento pertenece al conjunto. En caso afirmativo lo elimina.
- pertenece(conjunto, elemento). Comprueba si el elementos pertenece al conjunto. En caso afirmativo devuelve Verdadero.
- cardinal(conjunto). Devuelve la cantidad de elementos que posee el conjunto.

Implementación de un TAD en C

Las características del lenguaje que nos permiten implementar un TAD son: las estructuras, las funciones y los archivos de cabecera (.h).

Una estructura es una agrupación de campos (variables) de cualquier tipo predefinido, usuario o sistema, que guardan cierta relación lógica entre sí. Las estructuras nos permiten tratar a los campos que la componen como una sola unidad, como un todo.

Por ejemplo, si quisiéramos definir el TAD “Punto”, los datos a permiten representar un punto son sus coordenadas X e Y, Para esto podemos definir una estructura de la siguiente manera:

```
typedef struct {
   float x;
   float y;
} punto2D;

// definimos una variable de tipo punto2D
punto2D miPunto;
```

Archivos de cabecera (parte pública)

Los archivos de cabecera representan la parte pública de un TAD, su interfaz. Debemos pensar y definir correctamente las estructuras y operaciones antes de sentarnos a codificar ya que una vez que nuestro TAD esté en uso será muy difícil cambiar las interfaces.

Estos archivos los utilizaremos para agrupar declaraciones de estructuras de datos comunes y prototipos de funciones.

Definición	Prototipo de función: nos especifica el nombre de la función, los tipos de datos que recibe por parámetro y el tipo de dato que devuelve. Ejemplo:

	`punto2D crearPunto(int x, int y)`

Estos archivos deben ser incluídos en los archivos que contengan la implementación de las funciones y también en los archivos de código que hagan referencia a algún elemento aquí declarado (.c o .cpp).

Implementación de funciones (parte privada)

La implementación de funciones la realizaremos en un archivo con extensión .c o .cpp. Estos archivos representan la parte privada del TAD. En estos archivos se debe implementar cada una de las funciones declaradas en el archivo de cabecera definido previamente.

Por convención el archivo de cabecera y el archivo donde se realiza la implementación de las funciones deben tener el mismo nombre. Ejemplo:

- punto2D.h ➜ archivo de cabecera del TAD Punto
- punto2D.cpp ➜ archivo de implementación de funciones del TAD Punto

Aplicación de un TAD

A modo de resúmen, veremos un ejemplo simplificado de la distribución de código y archivos en la implementación y utilización de un TAD. Tomaremos como ejemplo el TAD punto.

punto2D.h

```
typedef struct {
   float x;
   float y;
} punto2D;

punto2D crearPunto(int,
int);

float distancia(punto2D,
punto2D);
```

punto2D.cpp

```
#include “punto2D.h”

punto2D crearPunto(int X,  int Y){
   punto2D punto;
   punto.x = x;
   punto.y = y;

   return punto;
}

float distancia(punto2D p1,  punto2D
p2){
   float d = sqrt( pow(p2.x-p1.x, 2) +
pow(p2.y-p1.y, 2) );

   return d;
}
```

main.cpp

```
#include “punto2D.h”

punto2D p1;
punto2D p2;

p1 = crearPunto(1,2);

p2 = crearPunto(1,2);

cout << distancia(p1,
p2) << endl;
```

Bibliografía

- Luis Joyanes Aguilar, Ignacio Zahonero Martínez. Programación en C. Segunda Edición. España: McGRAW-HILL/INTERAMERICANA DE ESPAÑA. S.A.U., 2005. ISBN-84: 481-9844-1.
- Brian W. Kernighan, Rob Pike. La práctica de la programación. Pearson Educación. Méjico (2000)
- Debugging with CodeBlocks: http://wiki.codeblocks.org/index.php?title=Debugging_with_Code::Blocks